# Design of Digital Circuits

## Lecture 12: Microarchitecture II

Prof. Onur Mutlu
ETH Zurich

Spring 2018

29 March 2019

# Talk Announcement – Monday, 1 April

- Monday, 1 April 2019, 10:30-11:30, CAB H52
- Towards Secure Integrated Circuit (IC) Fabrication: A Foundational Perspective on Hardware Security
- Prof. Siddharth Garg, New York University
  - https://safari.ethz.ch/siddharth-garg/

- Most semiconductor companies outsource IC fabrication to advanced external IC foundries. This is referred to as the "fabless" model. The fabless model comes at the expense of trust: Untrusted third-party foundries might overbuild and sell chips in the black market, or worse, maliciously modify the chip by inserting a "hardware Trojan". How can a designer protect from the twin threats of IP piracy and hardware Trojans?
- I will begin the talk by demonstrating the perils of heuristic security solutions by describing a powerful class of attacks (that we call SAT attacks) against state-of-the-art IP piracy defenses. I will then describe a well-founded approach to defending against SAT attacks using tools from cryptographic obfuscation. The second part of the talk will discuss provably secure defenses against hardware Trojans, this time by appealing foundational work in cryptography literature on verifiable computation.
- Full abstract and bio: https://safari.ethz.ch/siddharth-garg/

**Optional Review**

# Readings

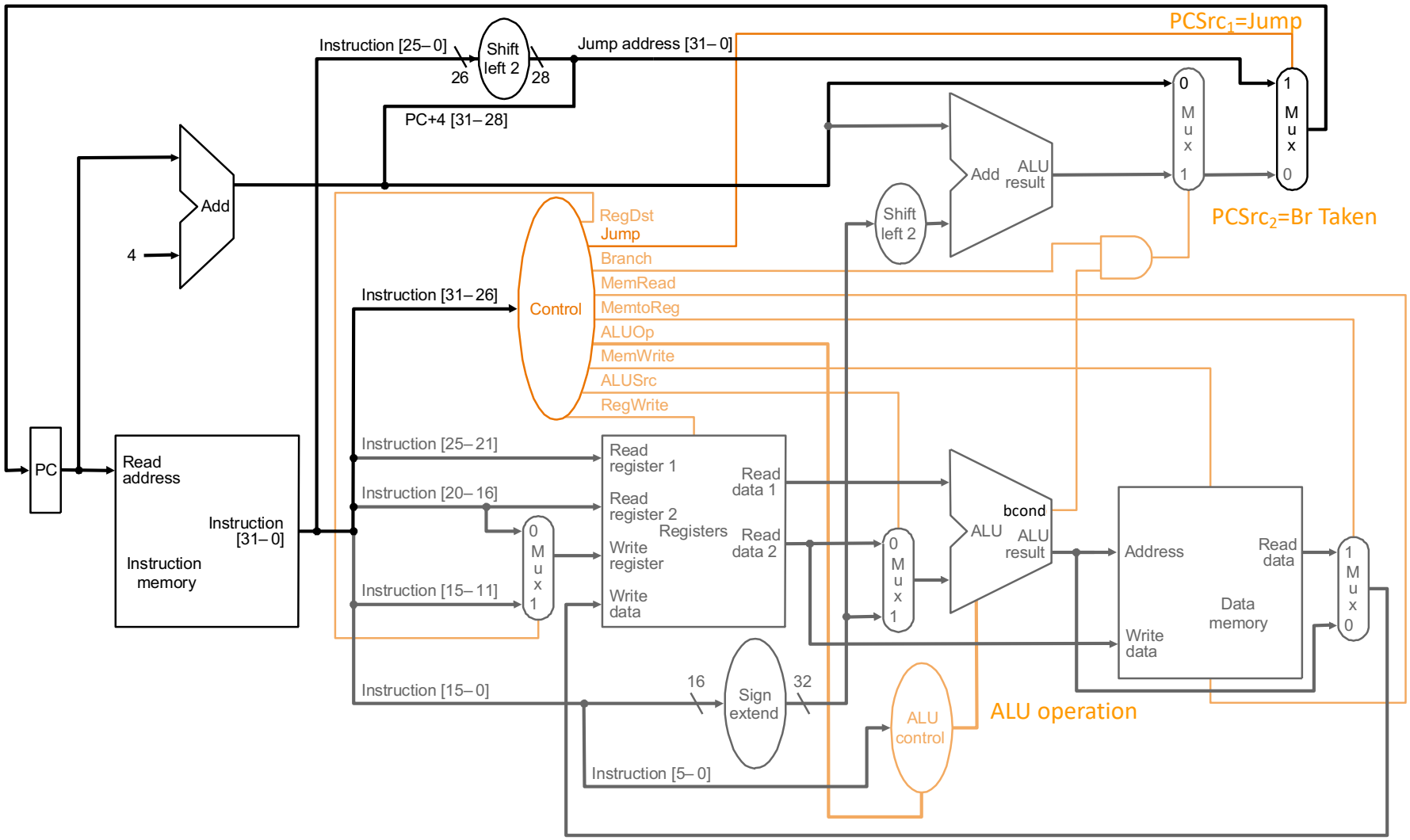
- This week
  - Introduction to microarchitecture and single-cycle microarchitecture
    - H&H, Chapter 7.1-7.3
    - P&P, Appendices A and C
  - Multi-cycle microarchitecture
    - H&H, Chapter 7.4
    - P&P, Appendices A and C

- Next week
  - Pipelining
    - H&H, Chapter 7.5
  - Pipelining Issues
    - H&H, Chapter 7.8.1-7.8.3

# Agenda for Today & Next Few Lectures

- Instruction Set Architectures (ISA): LC-3 and MIPS
- Assembly programming: LC-3 and MIPS
- Microarchitecture (principles & single-cycle uarch)
- Multi-cycle microarchitecture
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …
- Out-of-Order Execution

# Recall: Putting It All Together



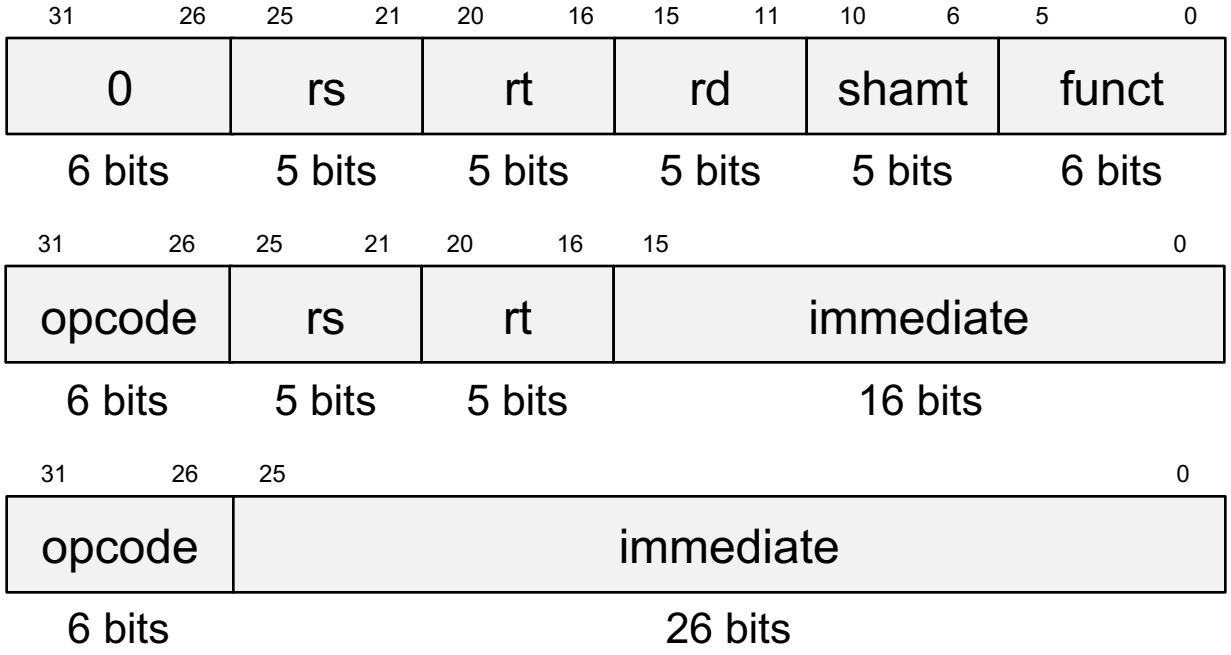

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted

# Single-Cycle Control Logic

# Recall: Single-Cycle Hardwired Control

- As combinational function of Inst=MEM[PC]




- Consider
  - All R-type and I-type ALU instructions
  - lw and sw
  - beq, bne, blez, bgtz
  - j, jr, jal, jalr

# Recall: Single-Bit Control Signals (I)

| | When De-asserted | When asserted | Equation |
|---|---|---|---|
| RegDest | GPR write select according to rt, i.e., inst[20:16] | GPR write select according to rd, i.e., inst[15:11] | opcode==0 |
| ALUSrc | 2nd ALU input from 2nd GPR read port | 2nd ALU input from sign-extended 16-bit immediate | (opcode!=0) && (opcode!=BEQ) && (opcode!=BNE) |
| MemtoReg | Steer ALU result to GPR write port | steer memory load to GPR write port | opcode==LW |
| RegWrite | GPR write disabled | GPR write enabled | (opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR)) |

JAL and JALR require additional RegDest and MemtoReg options

# Single-Bit Control Signals (II)

| | When De-asserted | When asserted | Equation |
|---|---|---|---|
| MemRead | Memory read disabled | Memory read port return load value | opcode==LW |
| MemWrite | Memory write disabled | Memory write enabled | opcode==SW |
| $PCSrc_1$ | According to $PCSrc_2$ | next PC is based on 26-bit immediate jump target | (opcode==J) \|\| (opcode==JAL) |
| $PCSrc_2$ | next PC = PC + 4 | next PC is based on 16-bit immediate branch target | (opcode==Bxx) && "bcond is satisfied" |

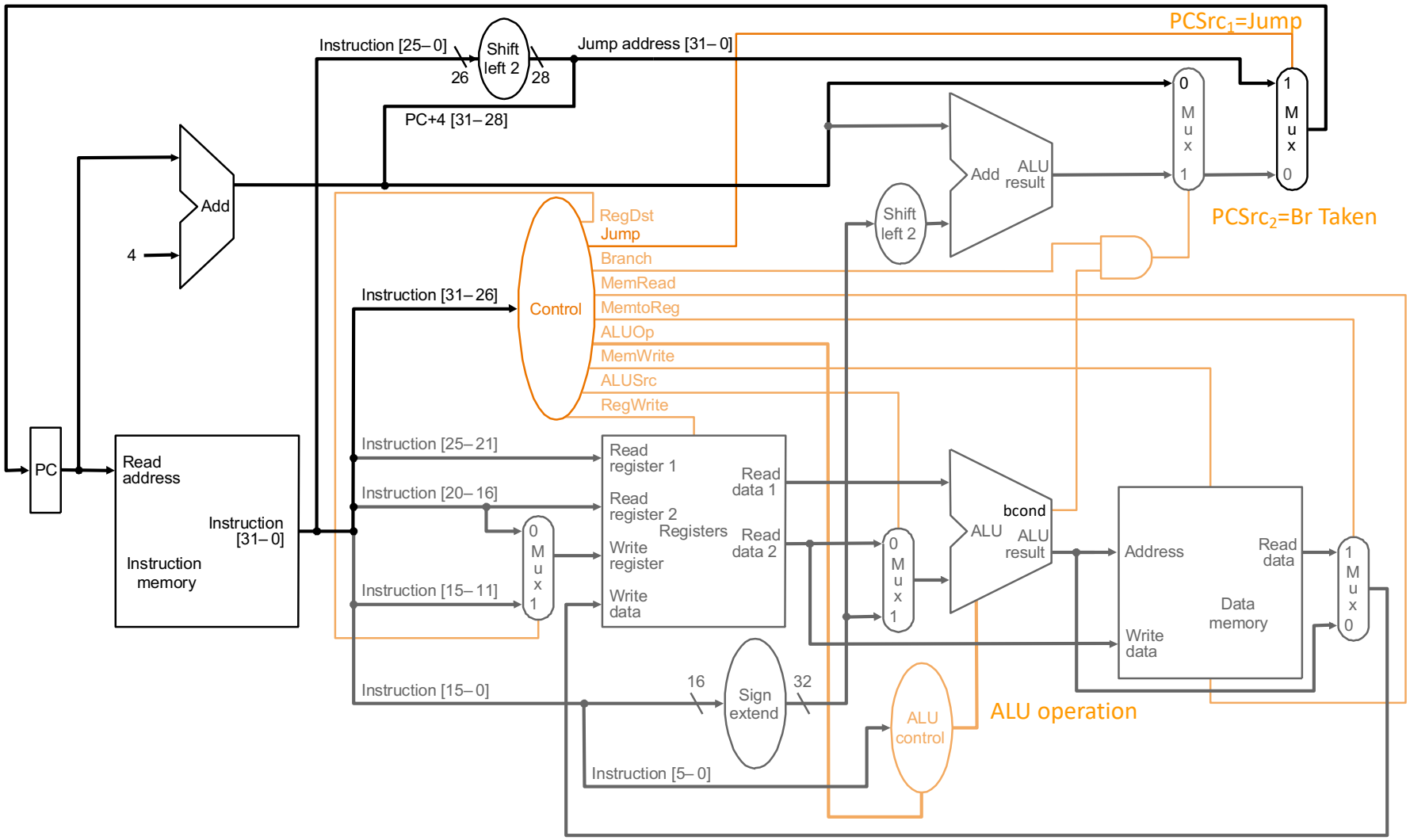
JR and JALR require additional PCSrc options

# ALU Control

- case opcode
  - '0' ⇒ select operation according to funct
  - 'ALUi' ⇒ selection operation according to opcode
  - 'LW' ⇒ select addition
  - 'SW' ⇒ select addition
  - 'Bxx' ⇒ select bcond generation function
  - __ ⇒ don't care

- Example ALU operations
  - ADD, SUB, AND, OR, XOR, NOR, etc.
  - bcond on equal, not equal, LE zero, GT zero, etc.

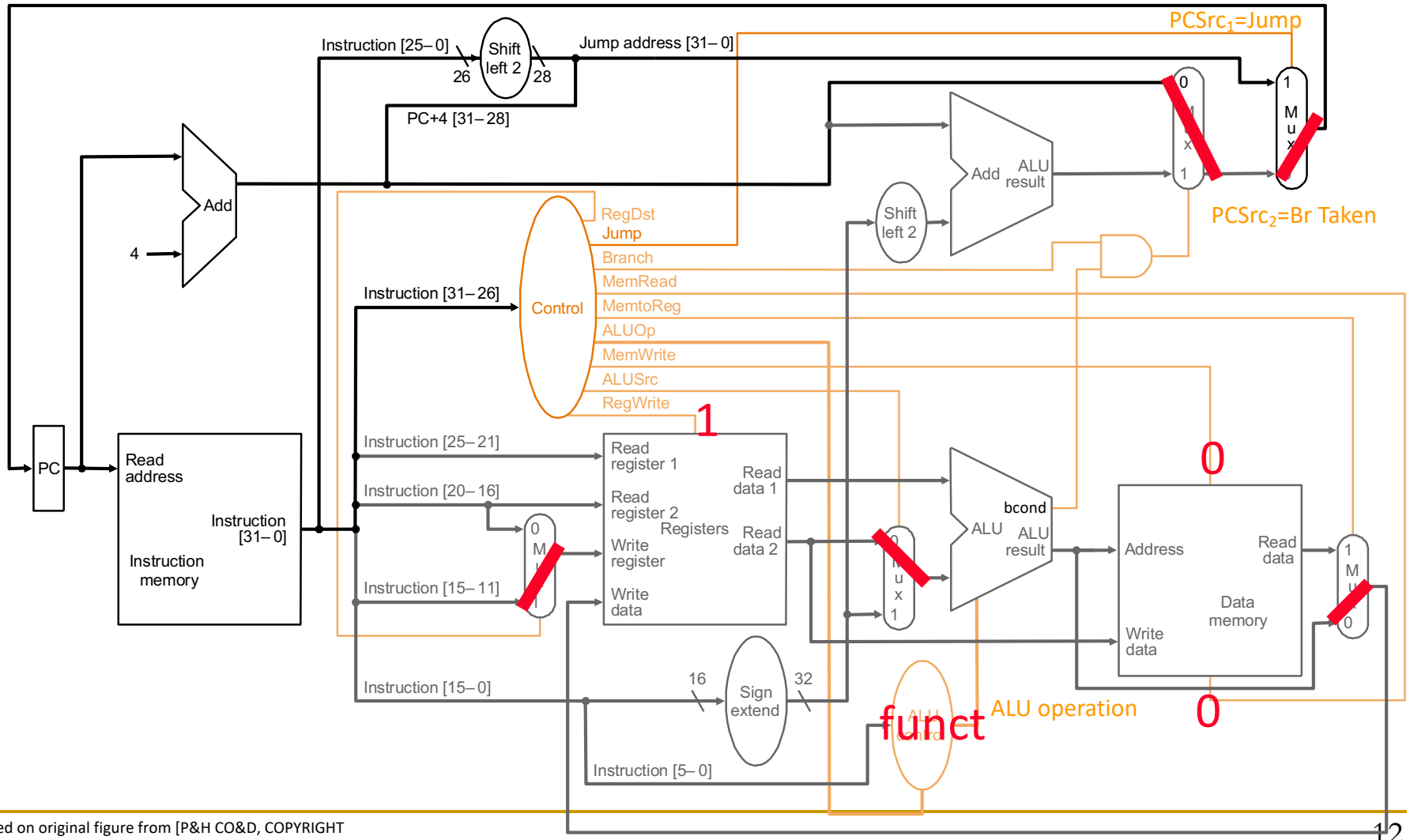
# Let's Control The Single-Cycle MIPS Datapath




**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

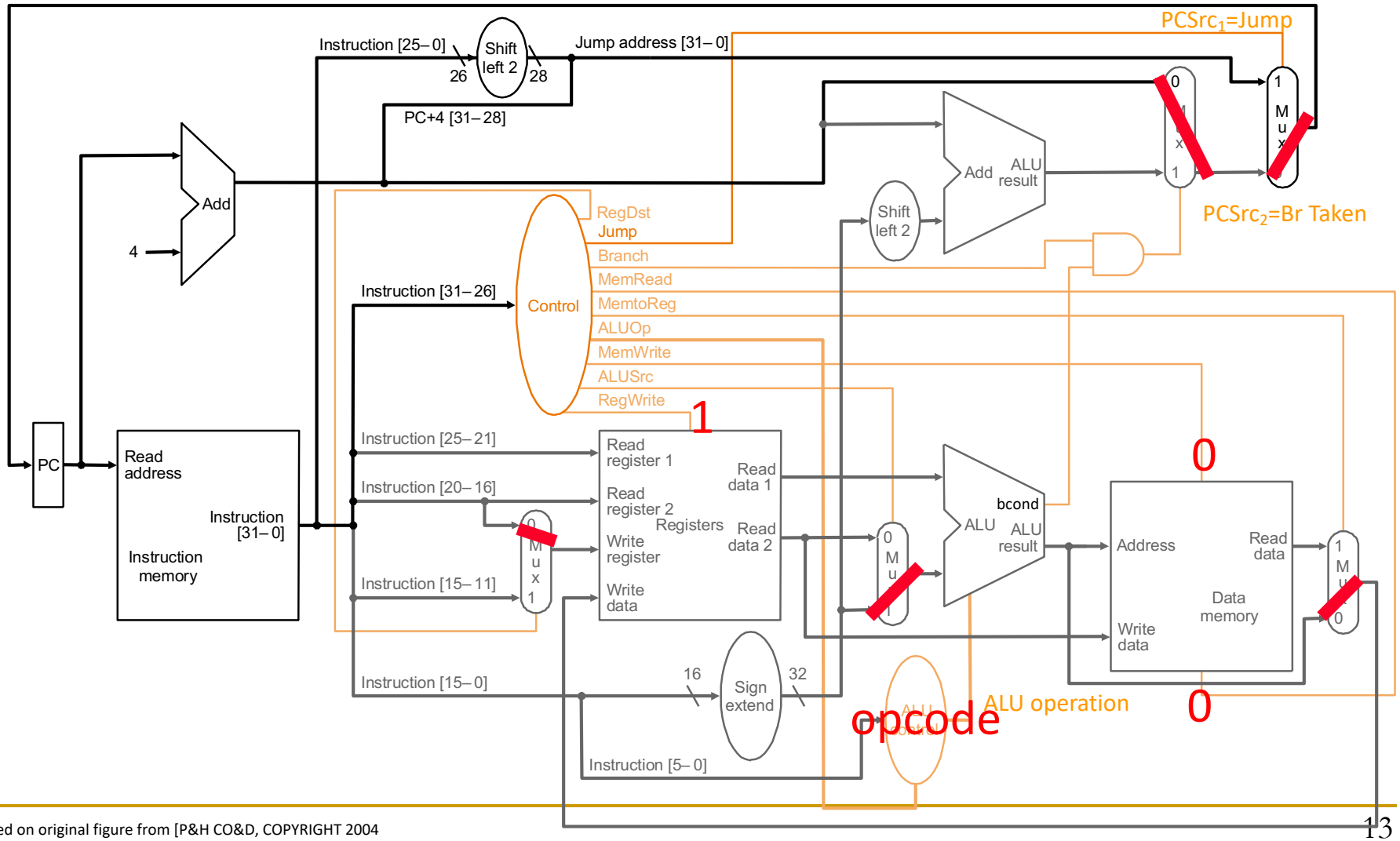
JAL, JR, JALR omitted

# R-Type ALU



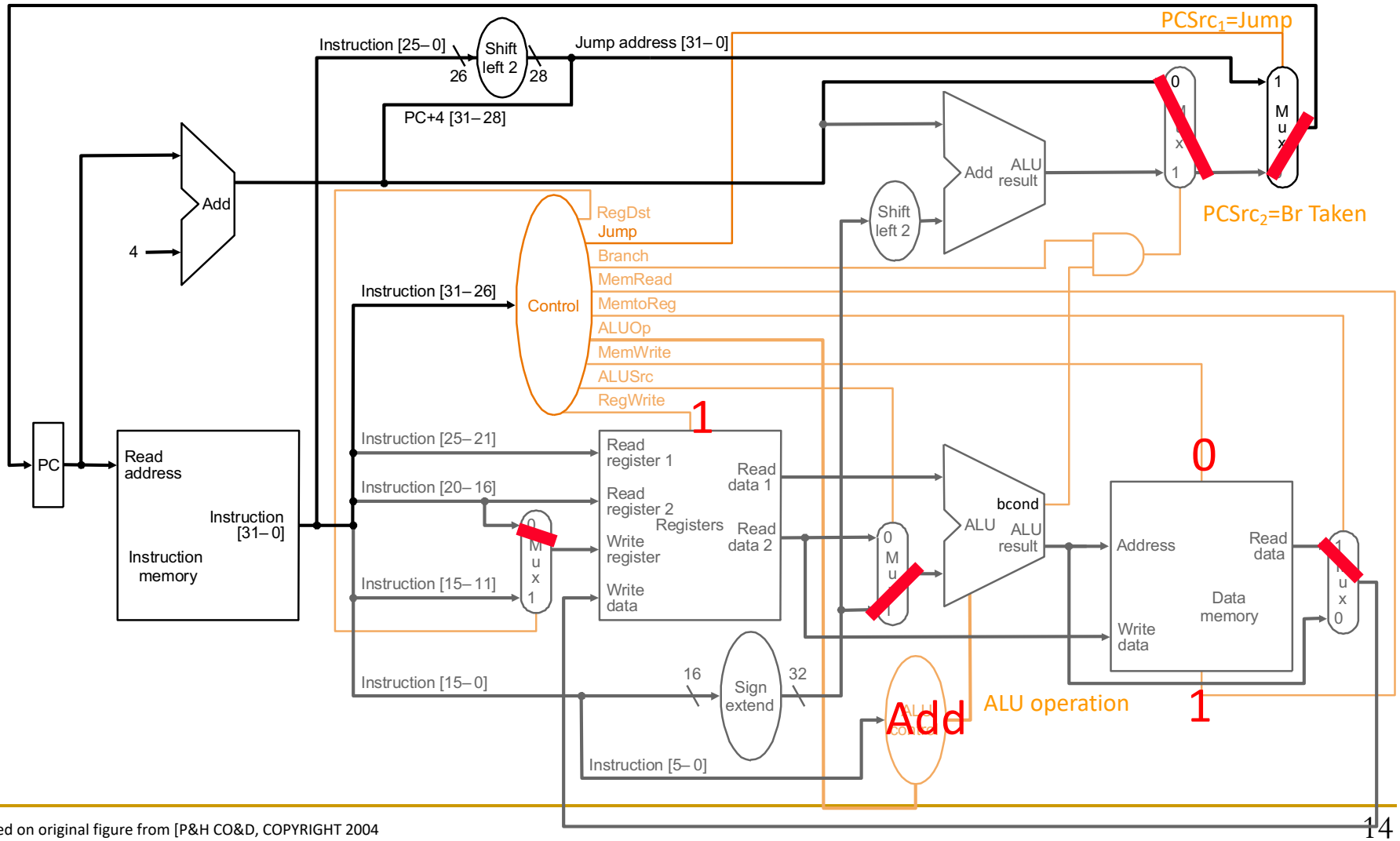

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# I-Type ALU



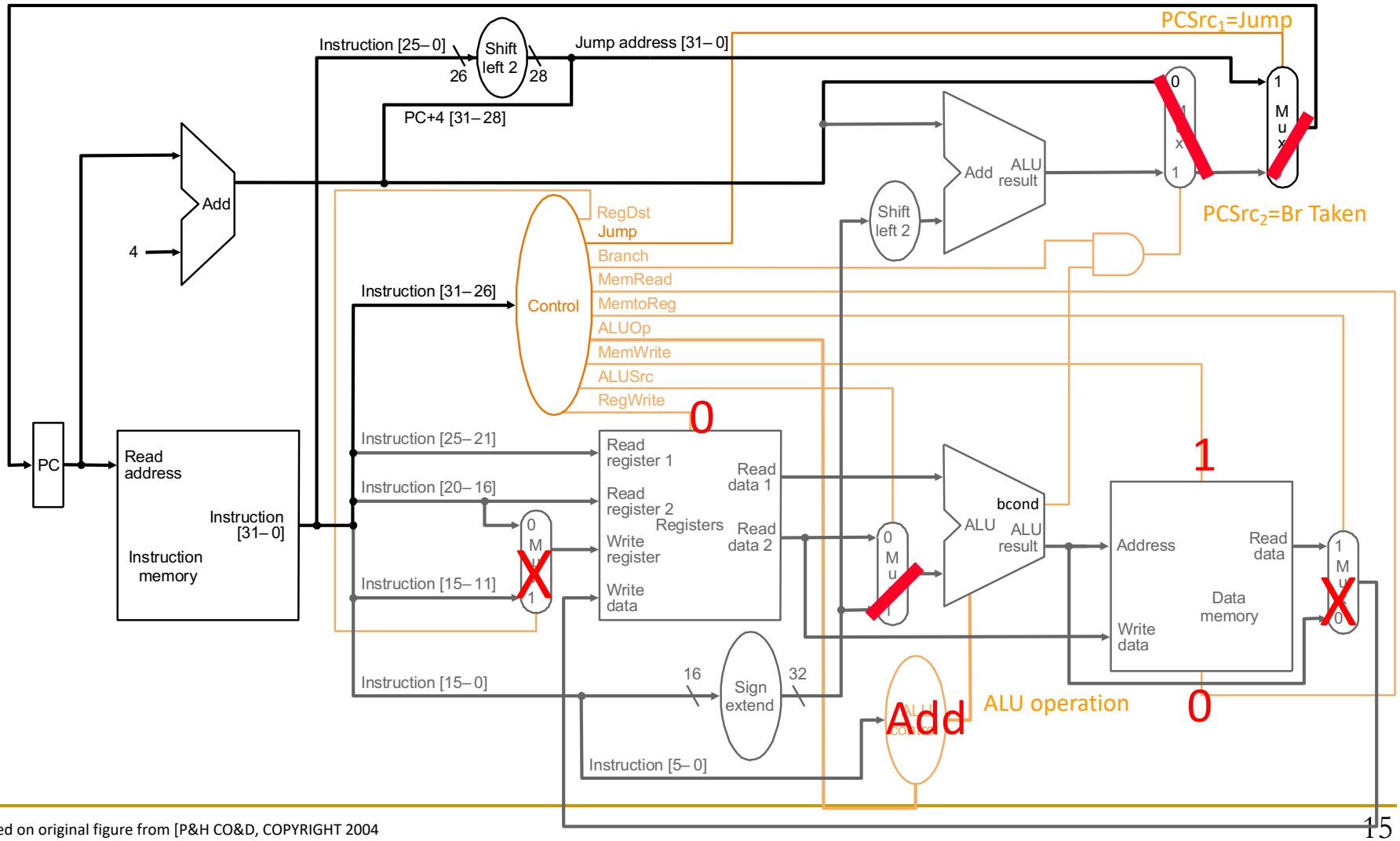

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# LW




**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

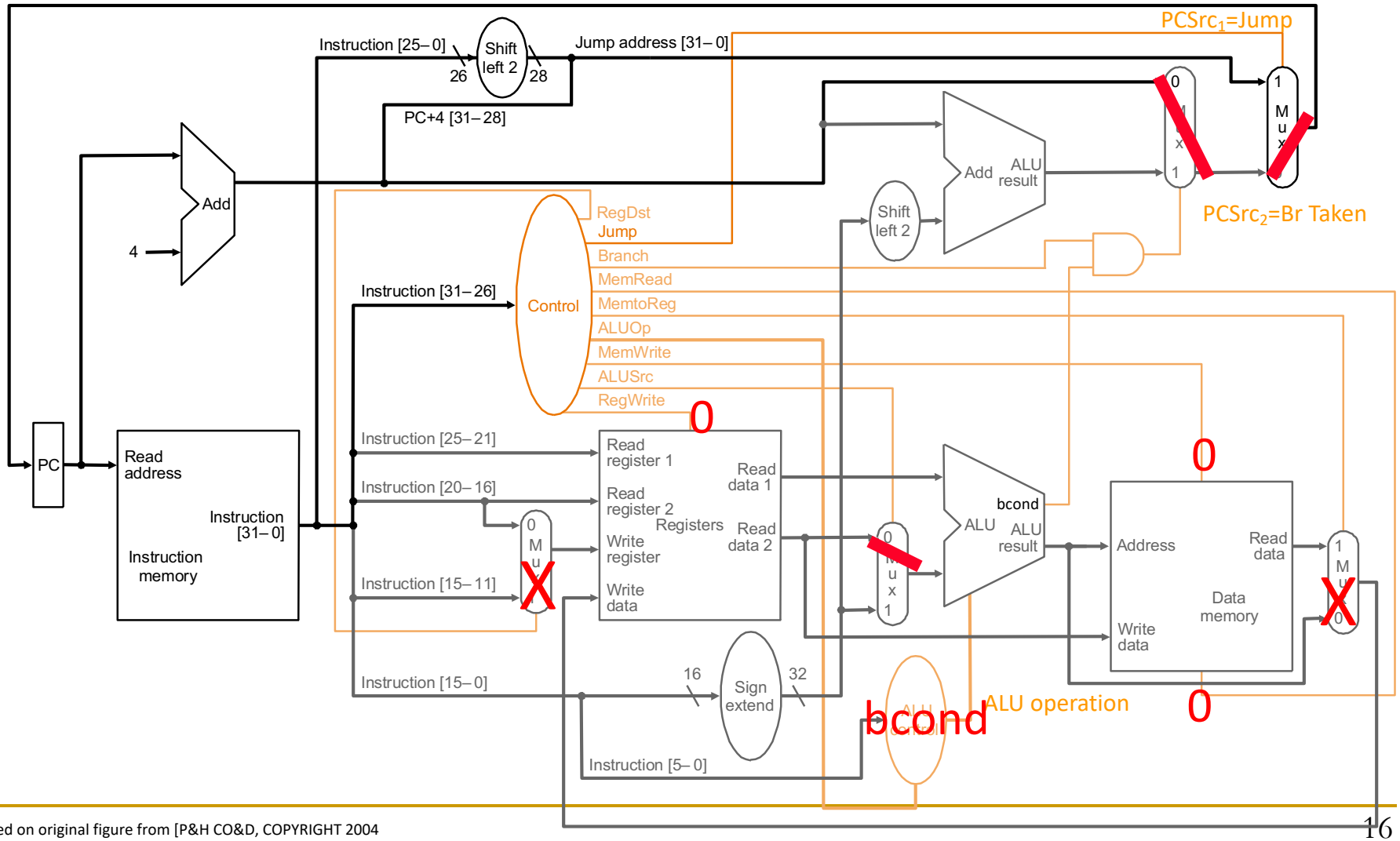
# SW




**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# Branch (Not Taken)

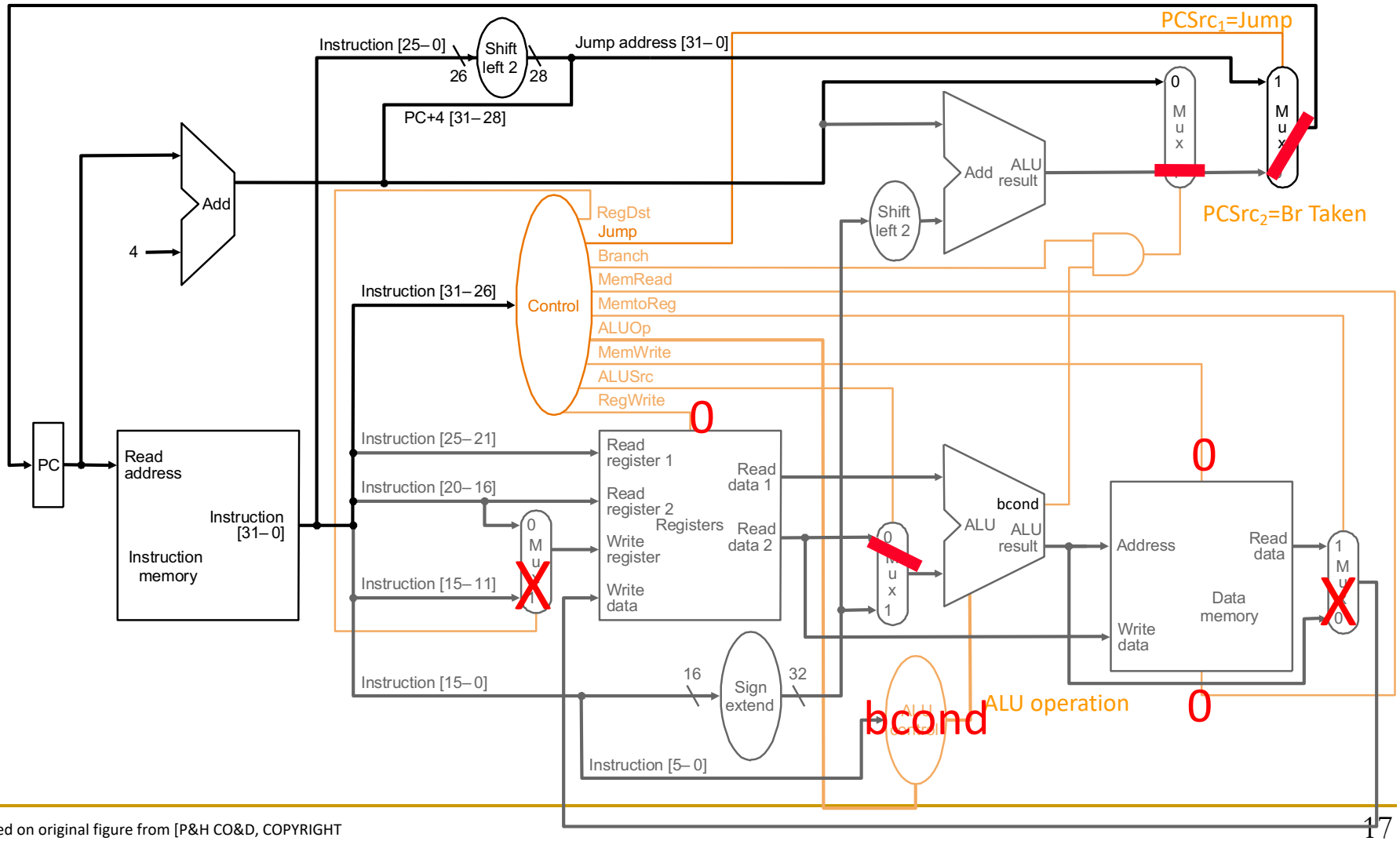
Some control signals are dependent on the processing of data



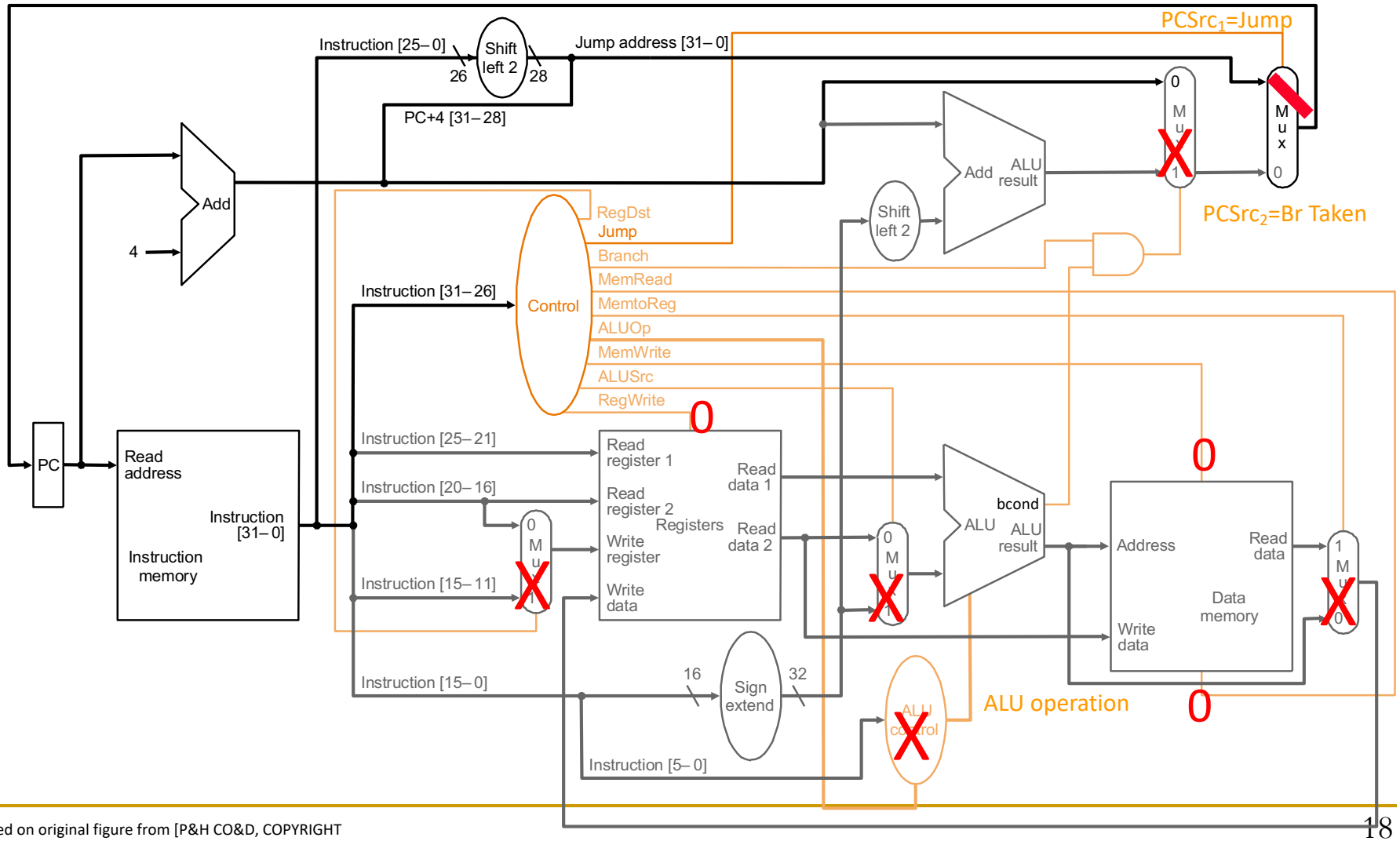

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# Branch (Taken)

Some control signals are dependent on the processing of data



# Jump



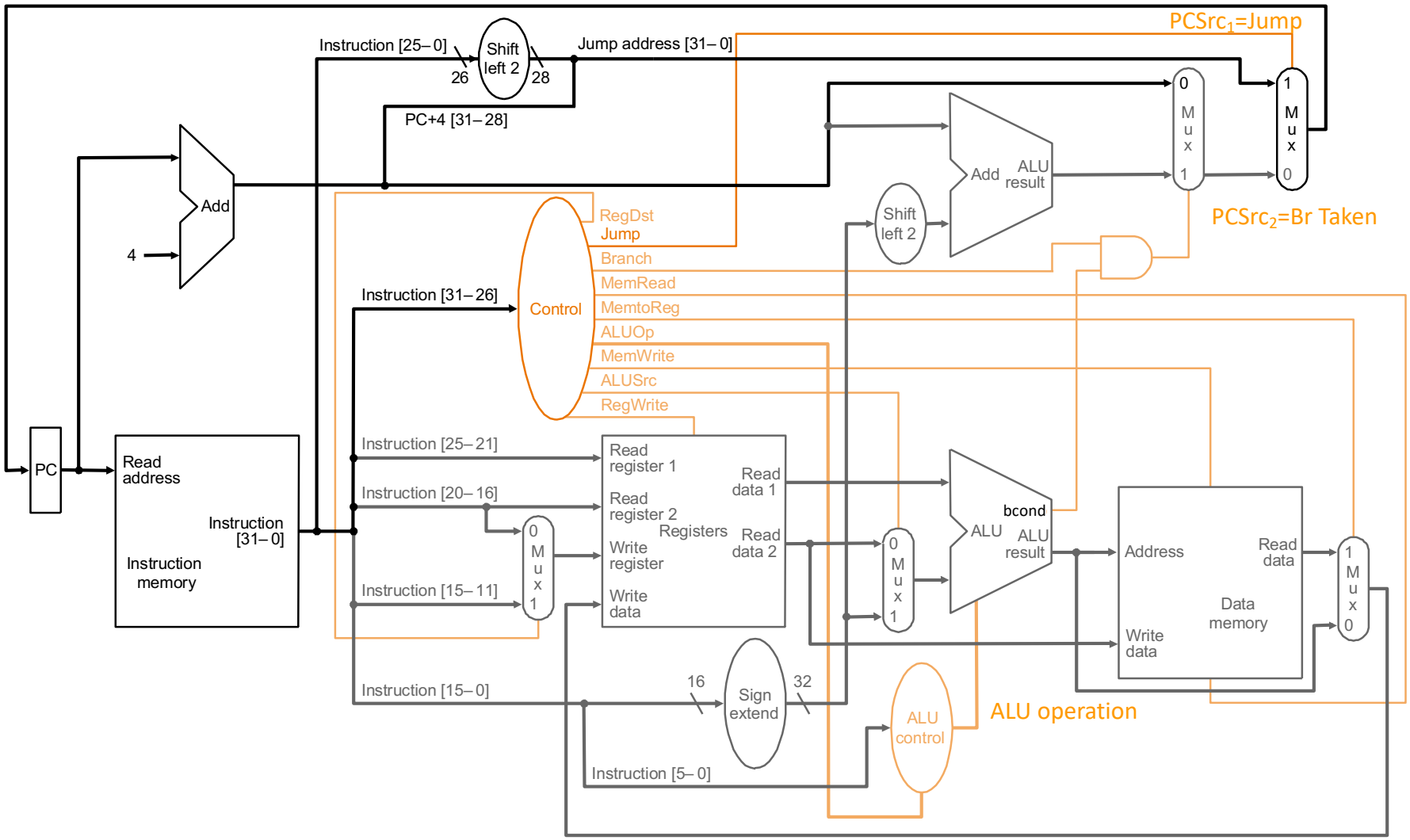

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# What is in That Control Box?

- Combinational Logic → Hardwired Control
  - Idea: Control signals generated combinationally based on instruction
  - Necessary in a single-cycle microarchitecture

- Sequential Logic → Sequential/Microprogrammed Control
  - Idea: A memory structure contains the control signals associated with an instruction
  - Control Store

# Review: Complete Single-Cycle Processor



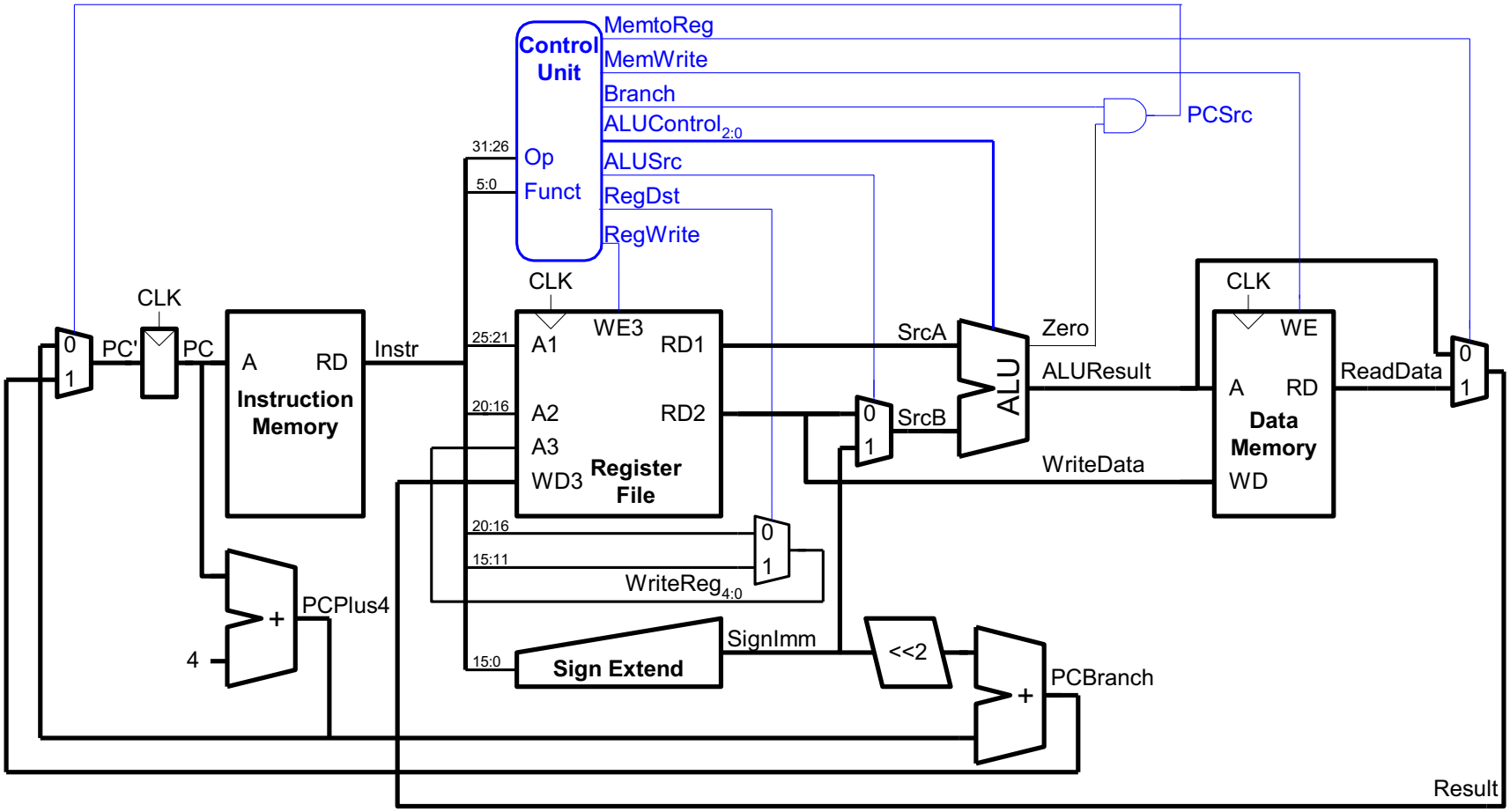

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted

# Another Single-Cycle MIPS Processor (from H&H)

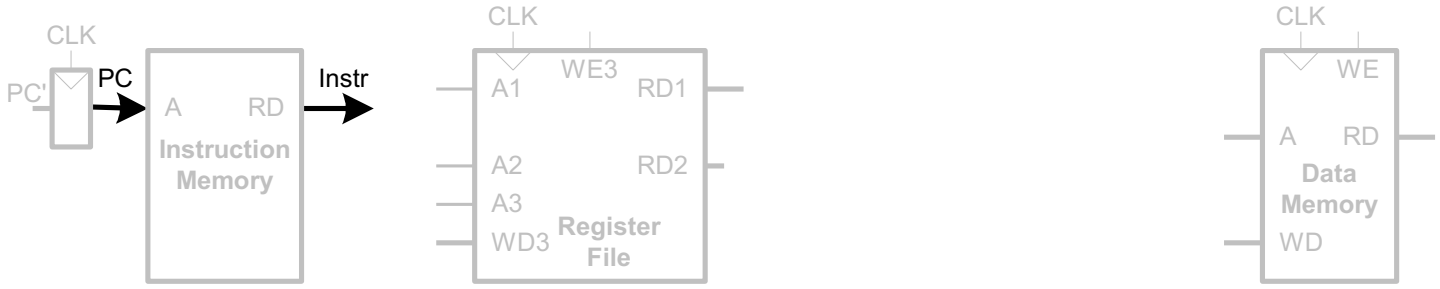
See backup slides to reinforce the concepts we have covered.
They are to complement your reading:
H&H, Chapter 7.1-7.3, 7.6

# Another Complete Single-Cycle Processor



# Example: Single-Cycle Datapath: lw fetch

- ***STEP 1:*** **Fetch instruction**

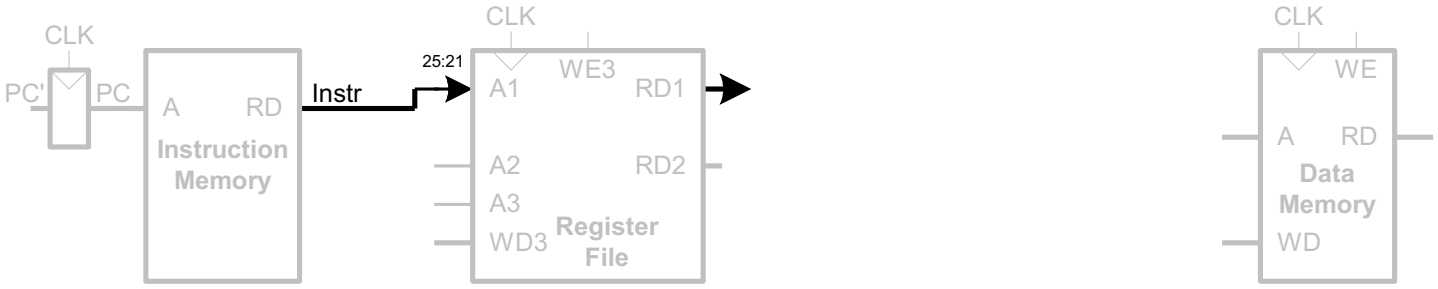



```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

# Single-Cycle Datapath: lw register read

- ***STEP 2:*** **Read source operands from register file**



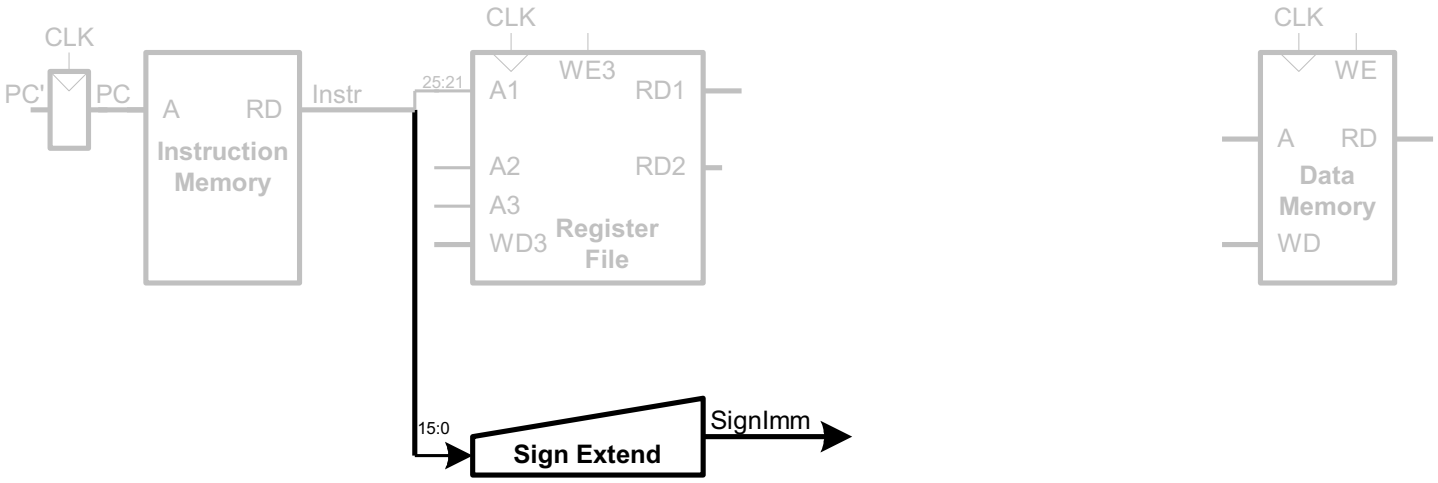

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: lw immediate

- ***STEP 3:*** **Sign-extend the immediate**



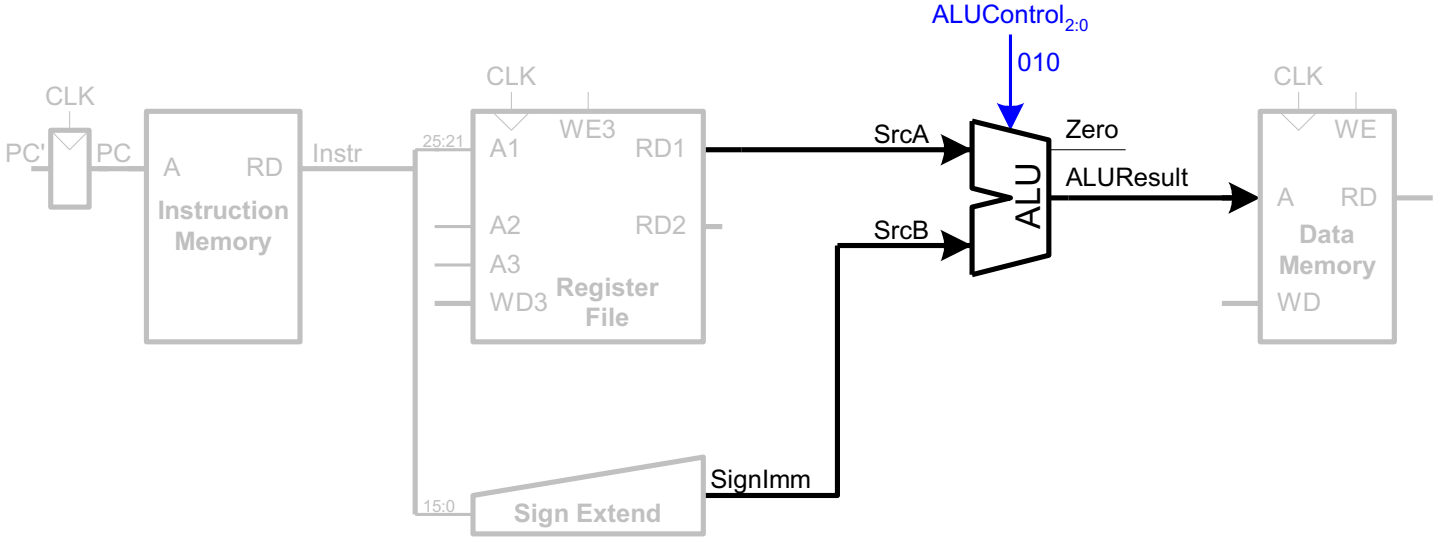

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: lw address

- ***STEP 4:*** **Compute the memory address**



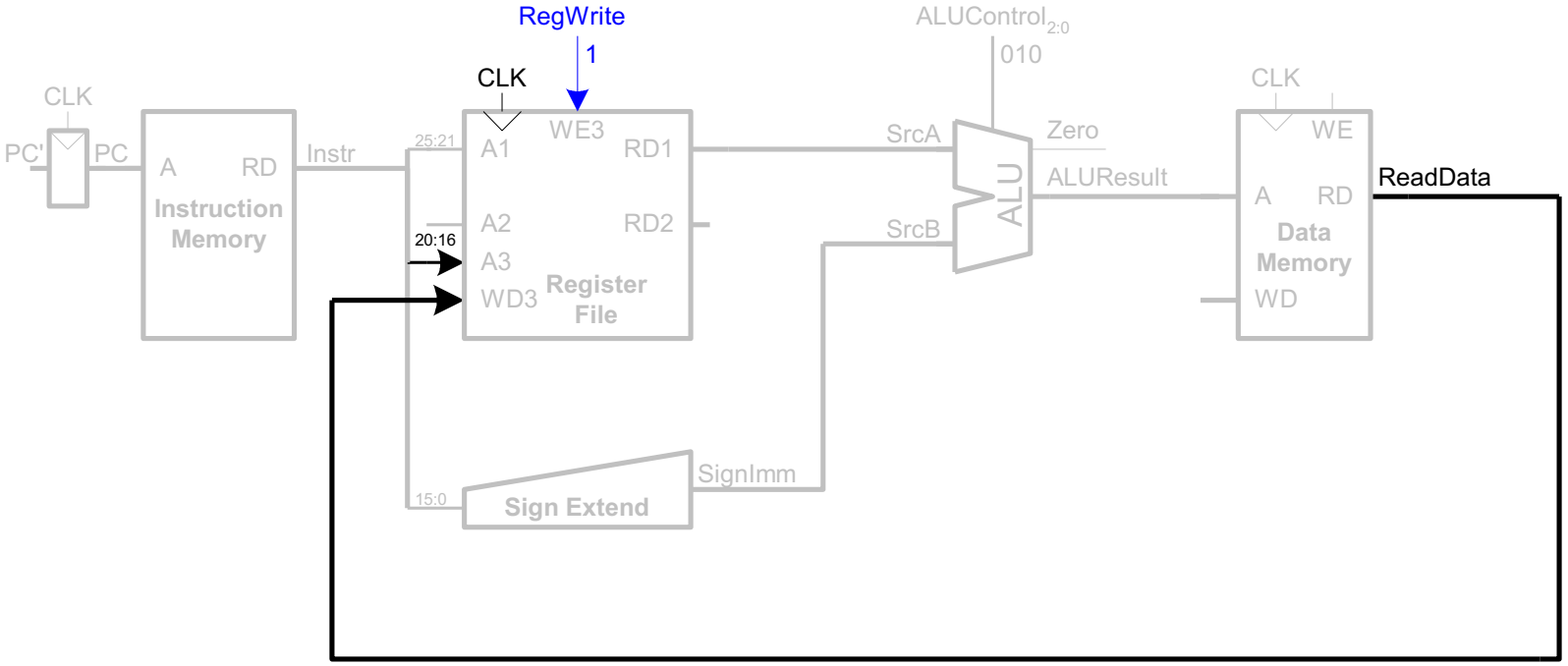

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: lw memory read

- ***STEP 5:*** **Read from memory and write back to register file**



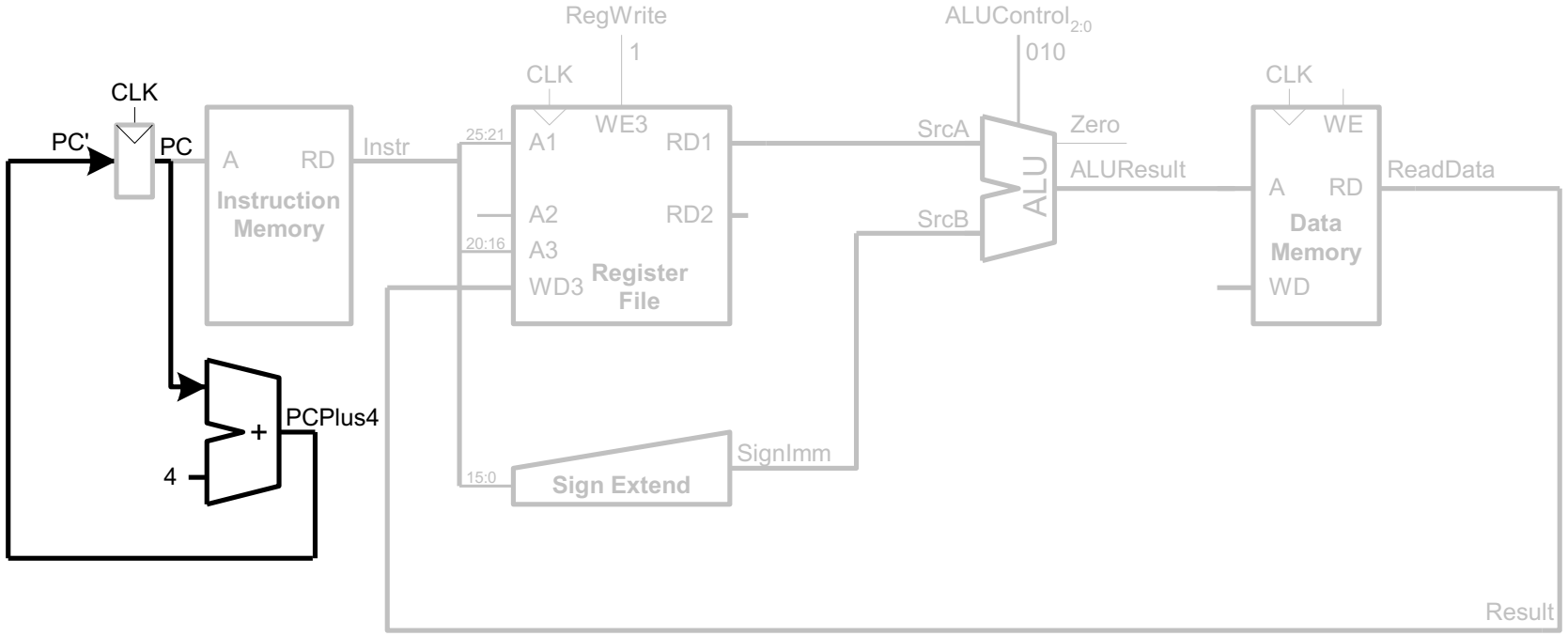

```
lw $s3, 1($0)   # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: lw PC increment

- ***STEP 6:*** **Determine address of next instruction**




```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

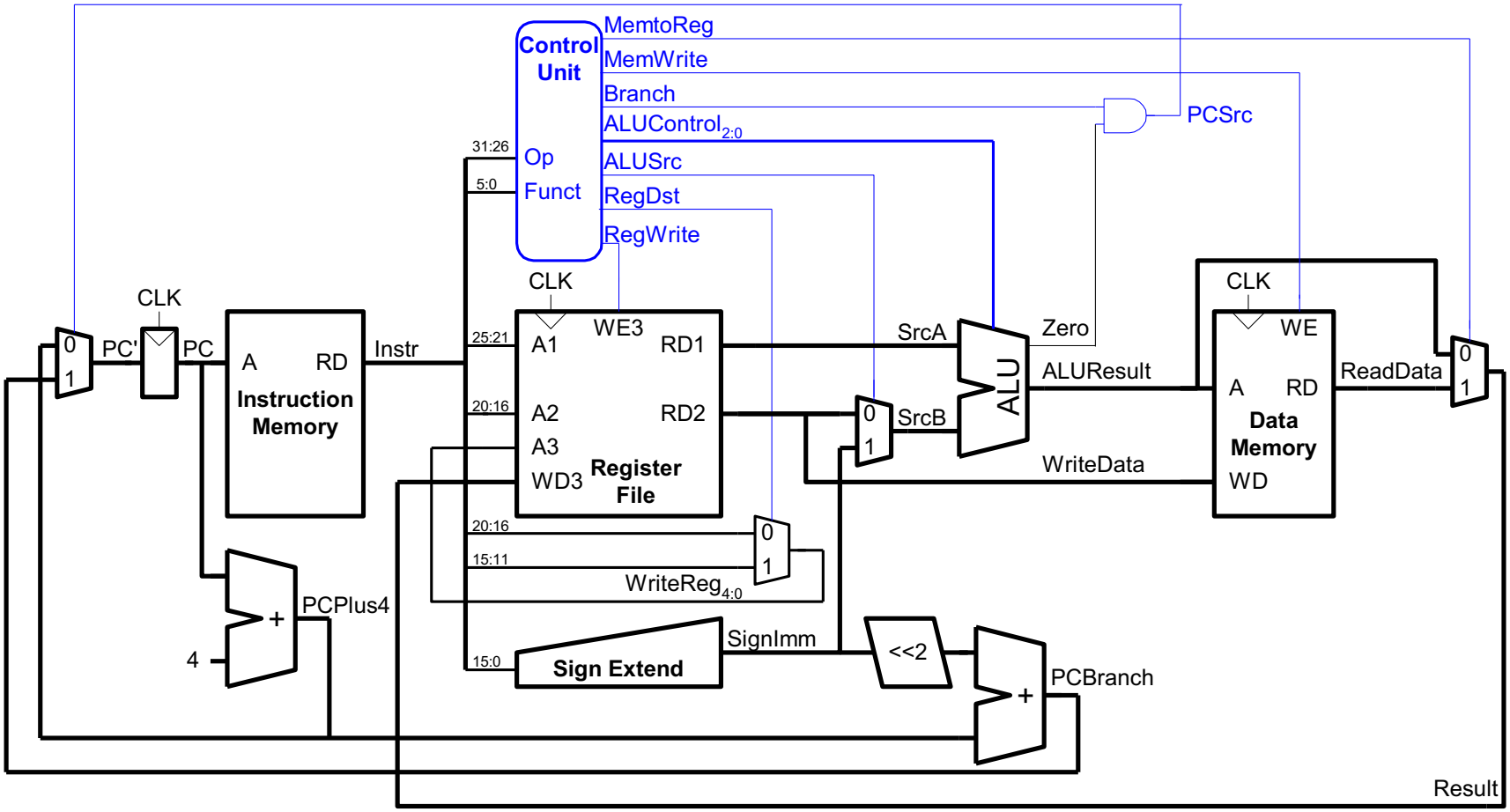
| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Similarly, We Need to Design the Control Unit

- Control signals generated by the decoder in control unit

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ | Jump |
|---|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 | 0 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 | 0 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 | 0 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 | 0 |
| addi | 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 00 | 0 |
| j | 000010 | 0 | X | X | X | 0 | X | XX | 1 |

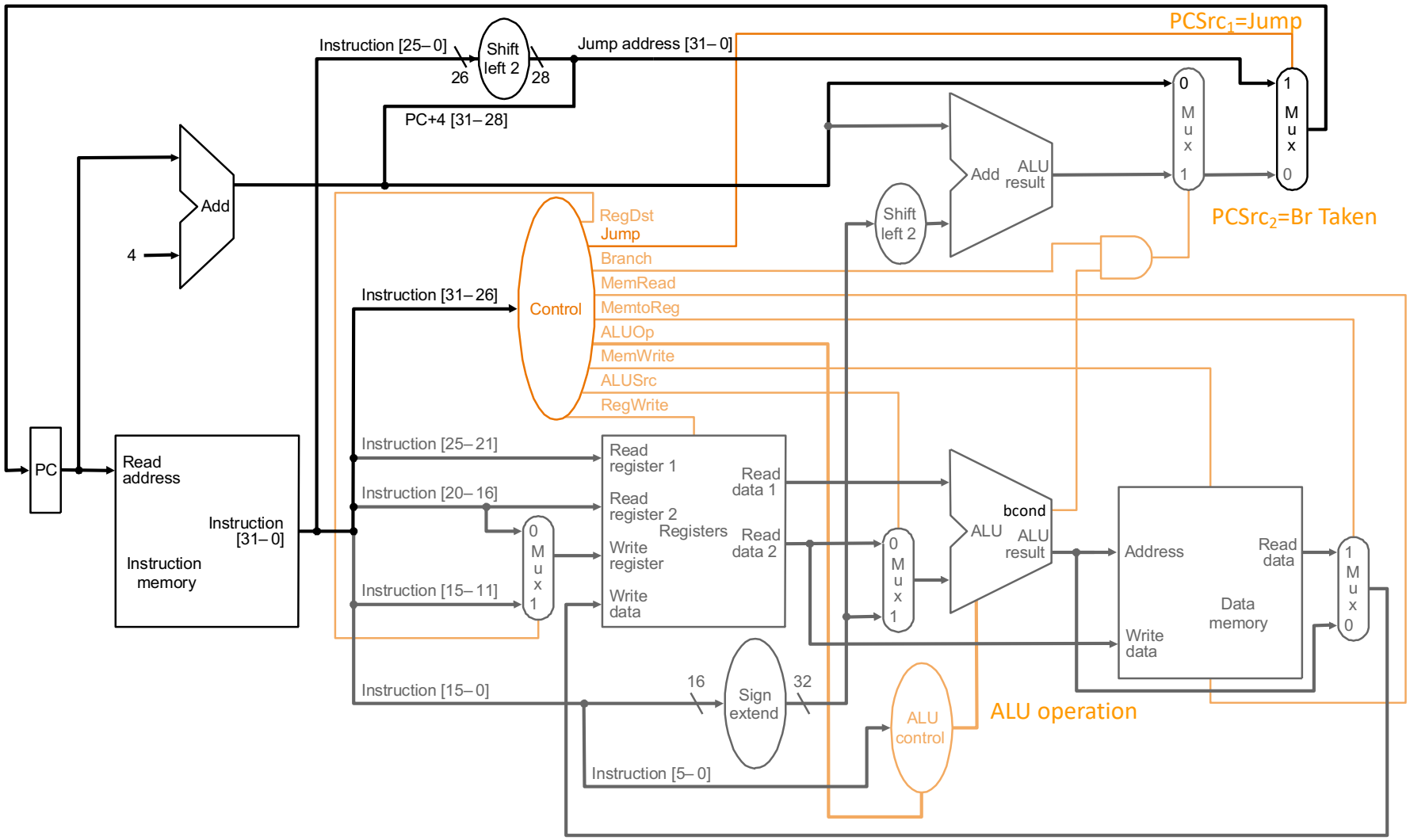
# Another Complete Single-Cycle Processor (H&H)



# Your Assignment

- Please read the Lecture Slides and the Backup Slides
- Please do your readings from the H&H Book
  - H&H, Chapter 7.1-7.3, 7.6

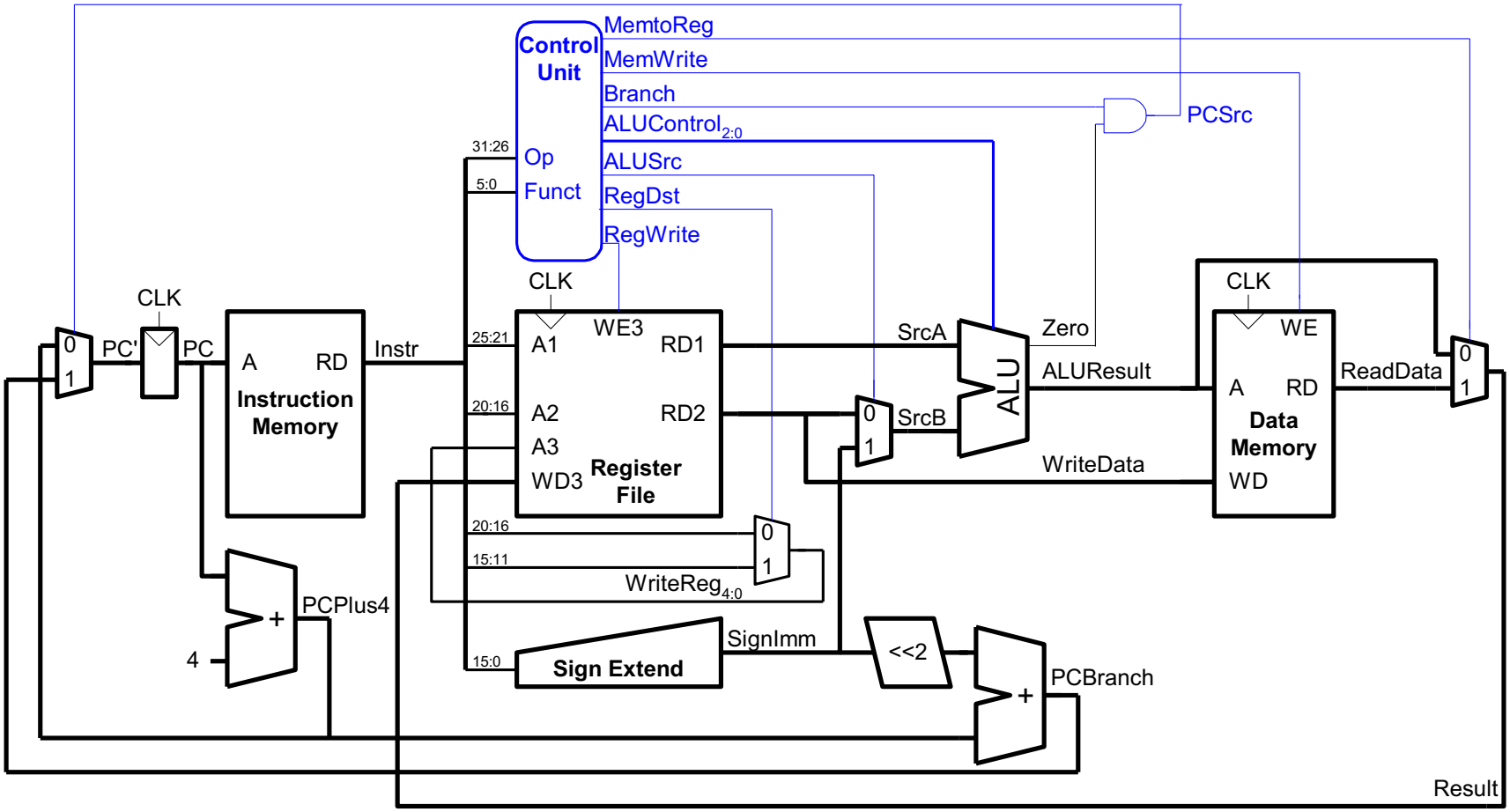
# Single-Cycle Uarch I (We Developed in Lectures)




**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted

# Single-Cycle Uarch II (In Your Readings)



# Evaluating the Single-Cycle Microarchitecture

# A Single-Cycle Microarchitecture

- Is *this* a good idea/design?
- When is this a good design?
- When is this a bad design?
- How can we design a better microarchitecture?

# Performance Analysis Basics

# Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

# Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.
- **So how fast are my instructions ?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

# Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.
- **So how fast are my instructions ?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***
- **How much time is one clock cycle?**
  - The critical path determines how much time one cycle requires = ***clock period***.
  - 1/clock period = ***clock frequency*** = how many cycles can be done each second.

# Processor Performance

- **Now as a general formula**
  - Our program consists of executing **N** instructions.
  - Our processor needs **CPI** cycles for each instruction.
  - The maximum clock speed of the processor is **f**, and the clock period is therefore **T**=1/f

# Processor Performance

- **Now as a general formula**
  - Our program consists of executing **N** instructions.
  - Our processor needs **CPI** cycles for each instruction.
  - The maximum clock speed of the processor is **f**, and the clock period is therefore **T**=1/f
- **Our program executes in**

**N x CPI x (1/f) =**

**N x CPI x T seconds**

# Performance Analysis Basics

- Execution time of an instruction
  - {CPI} x {clock cycle time}
    - CPI: Number of cycles it takes to execute an instruction

- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

# Performance Analysis of Our Single-Cycle Design

# A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

# What is the Slowest Instruction to Process?

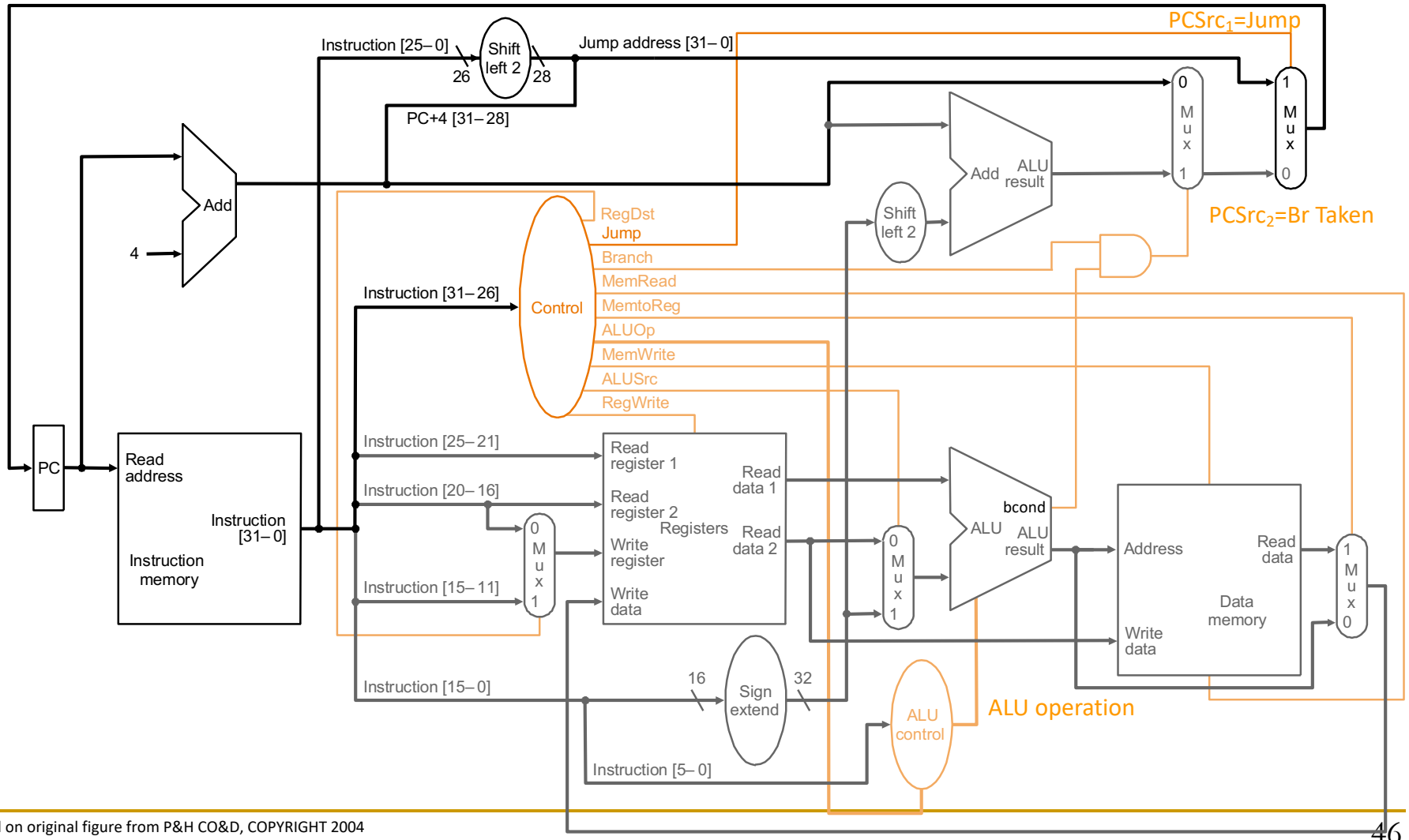
- Let's go back to the basics

- All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

- Do each of the above phases take the same time (latency) for all instructions?

# Let's Find the Critical Path

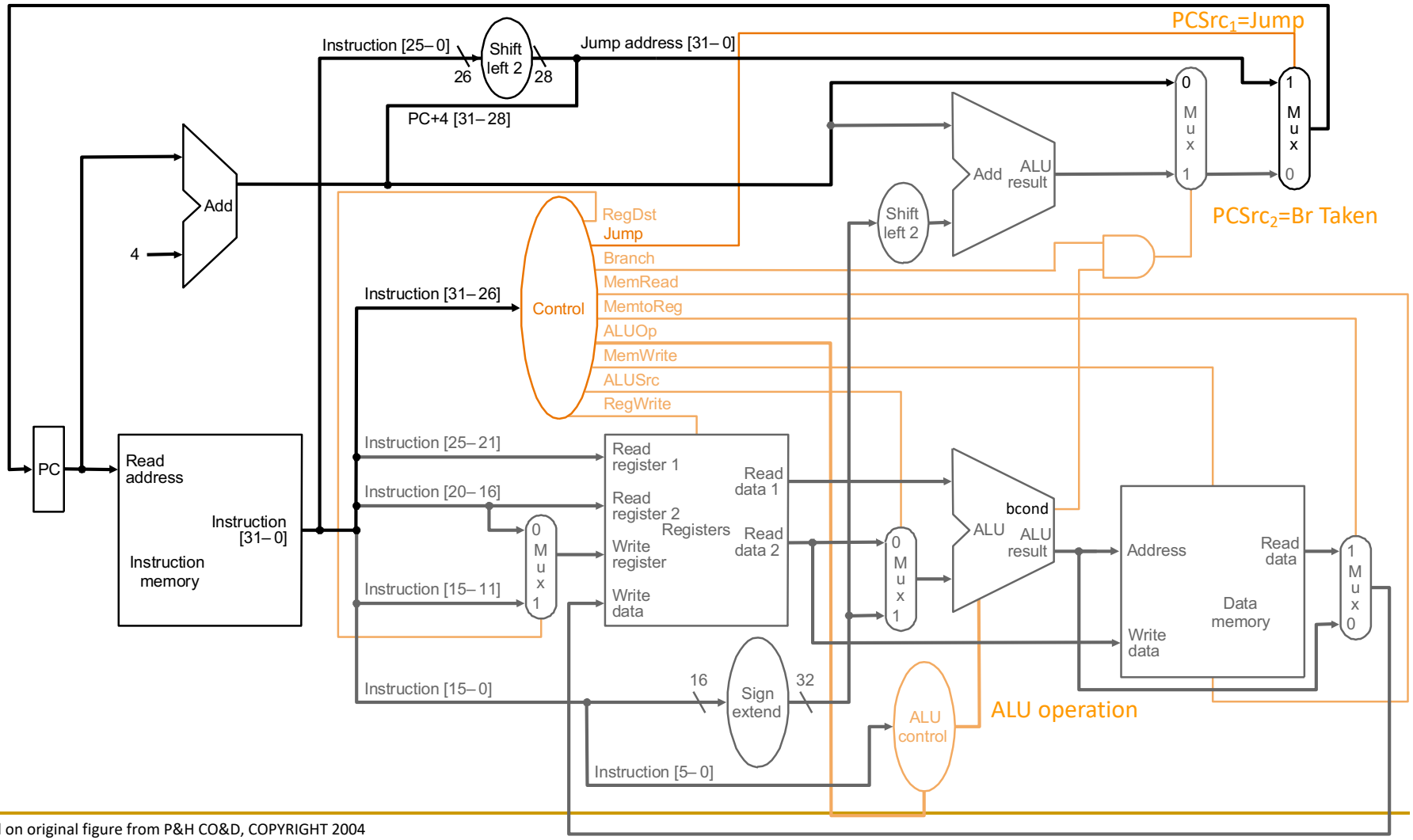


# Example Single-Cycle Datapath Analysis

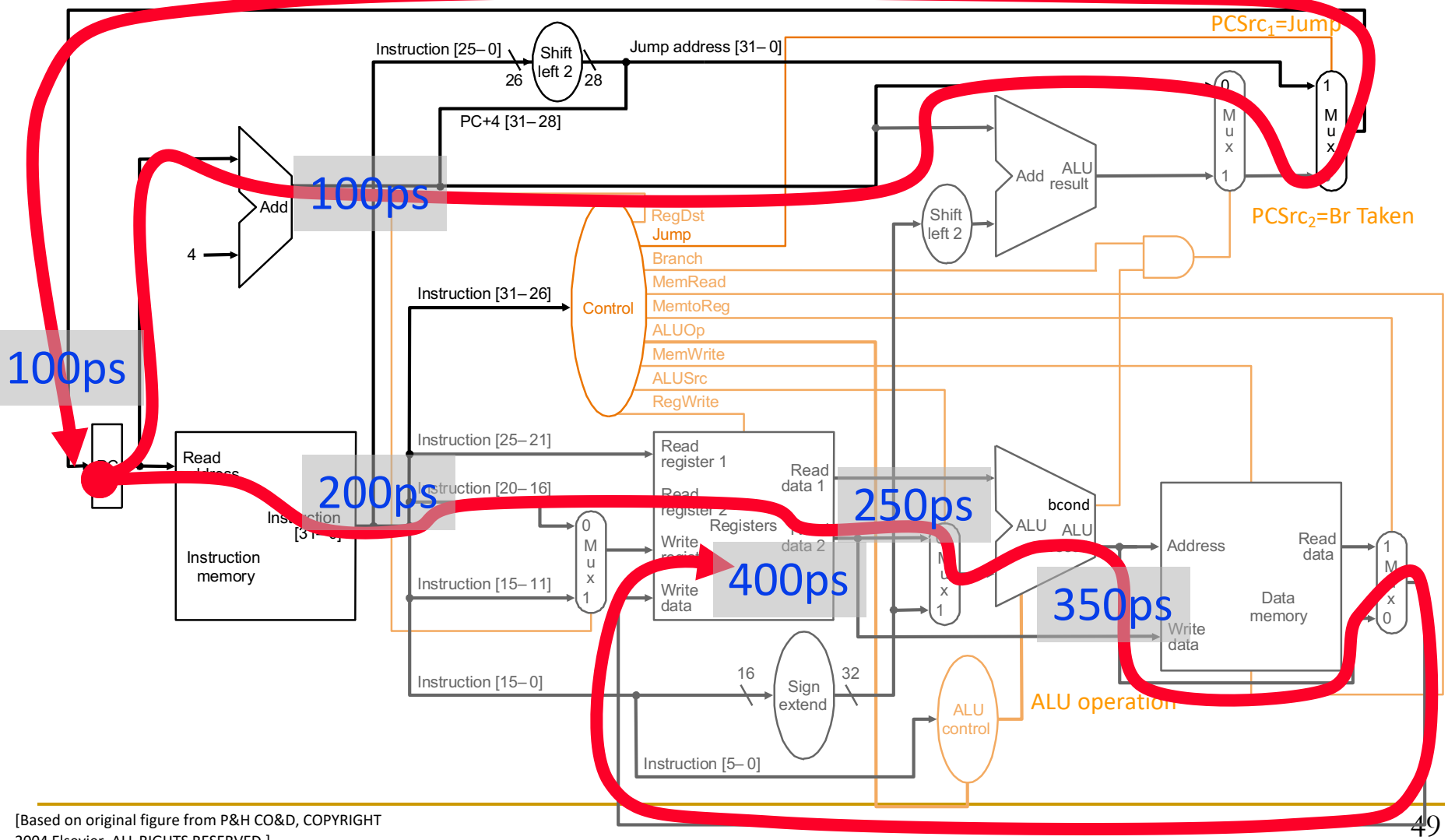
- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

| steps | IF | ID | EX | MEM | WB | Delay |
|---|---|---|---|---|---|---|
| resources | mem | RF | ALU | mem | RF | |
| R-type | 200 | 50 | 100 | | 50 | 400 |
| I-type | 200 | 50 | 100 | | 50 | 400 |
| LW | 200 | 50 | 100 | 200 | 50 | 600 |
| SW | 200 | 50 | 100 | 200 | | 550 |
| Branch | 200 | 50 | 100 | | | 350 |
| Jump | 200 | | | | | 200 |

# Let's Find the Critical Path

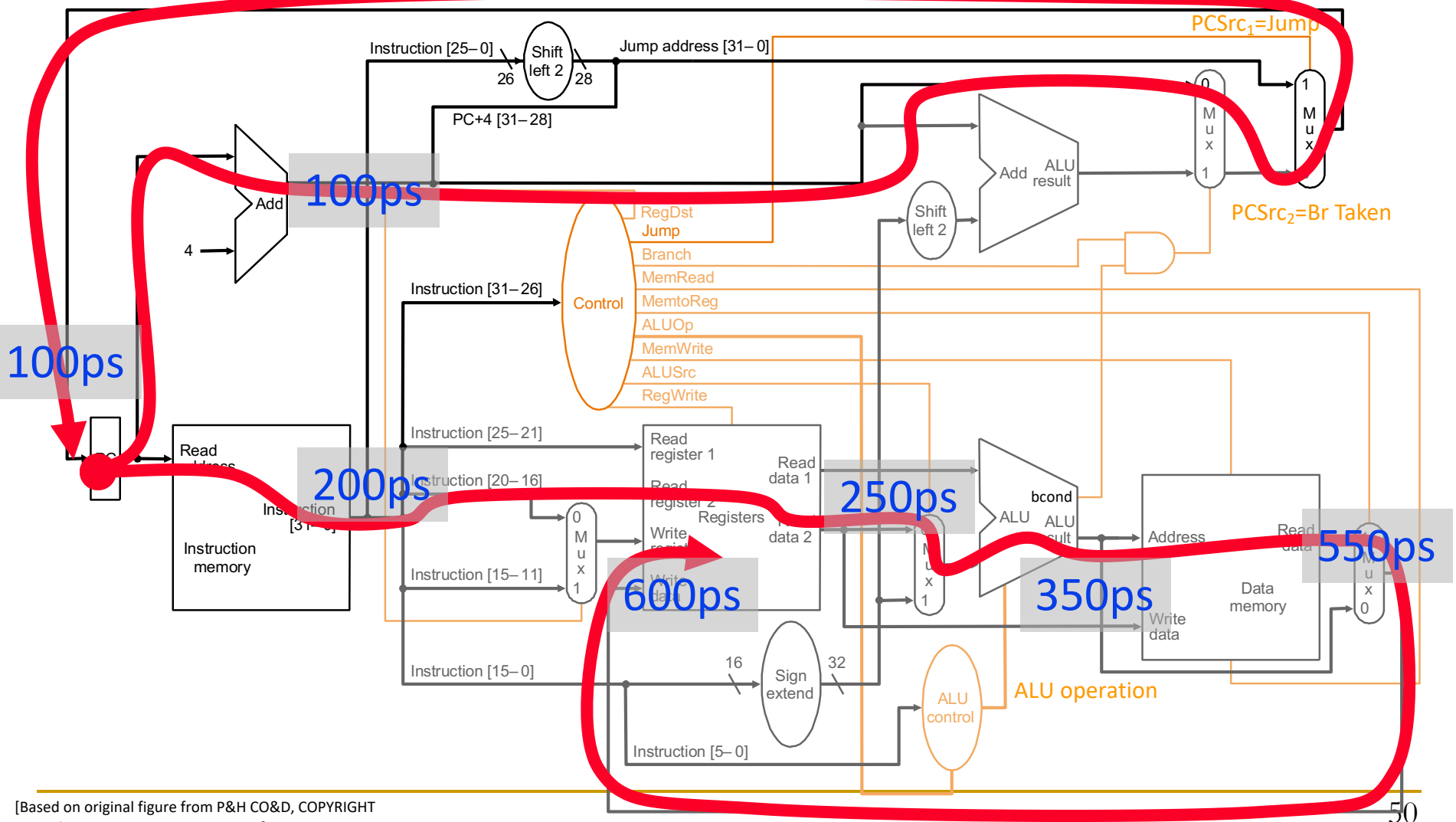


# R-Type and I-Type ALU

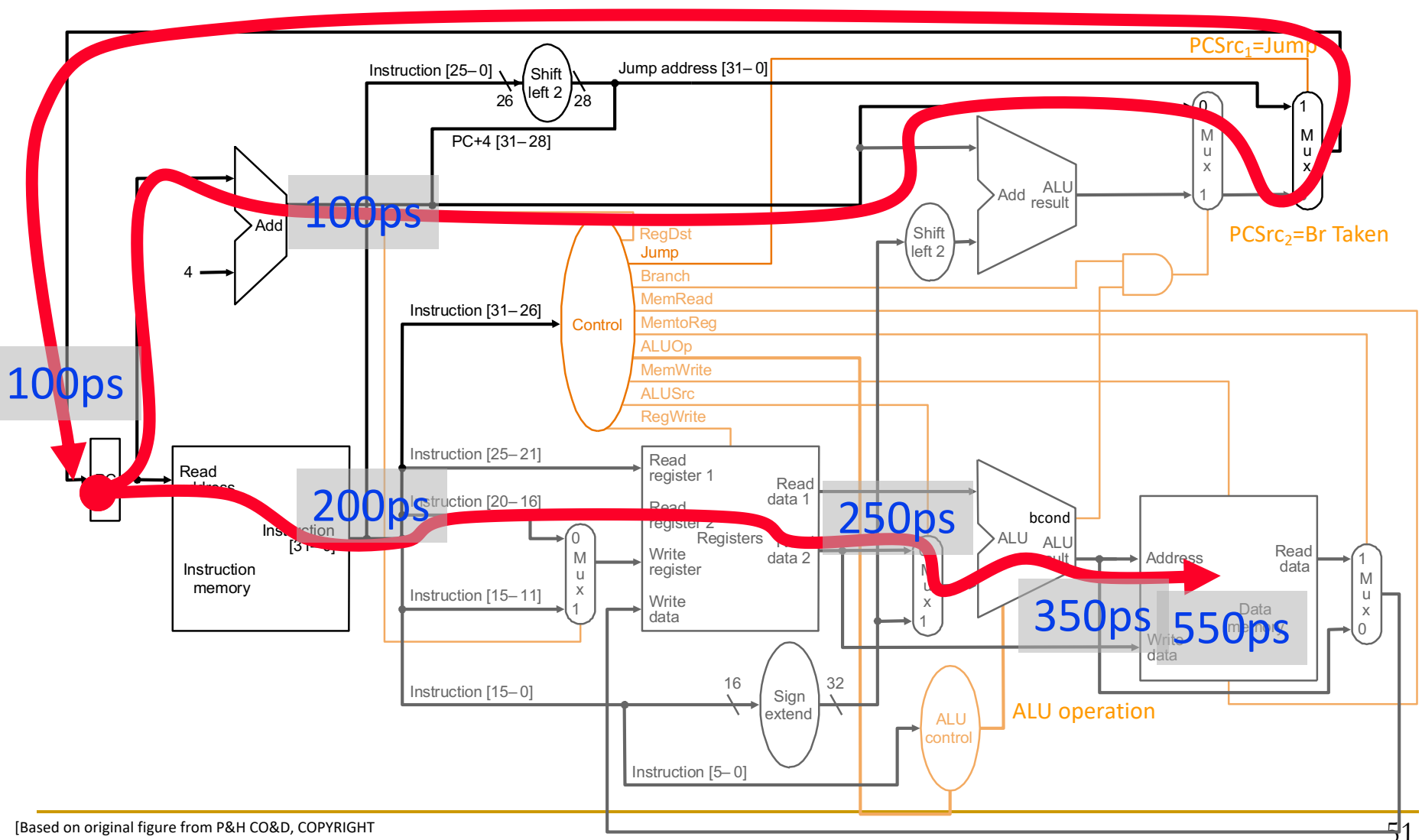



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

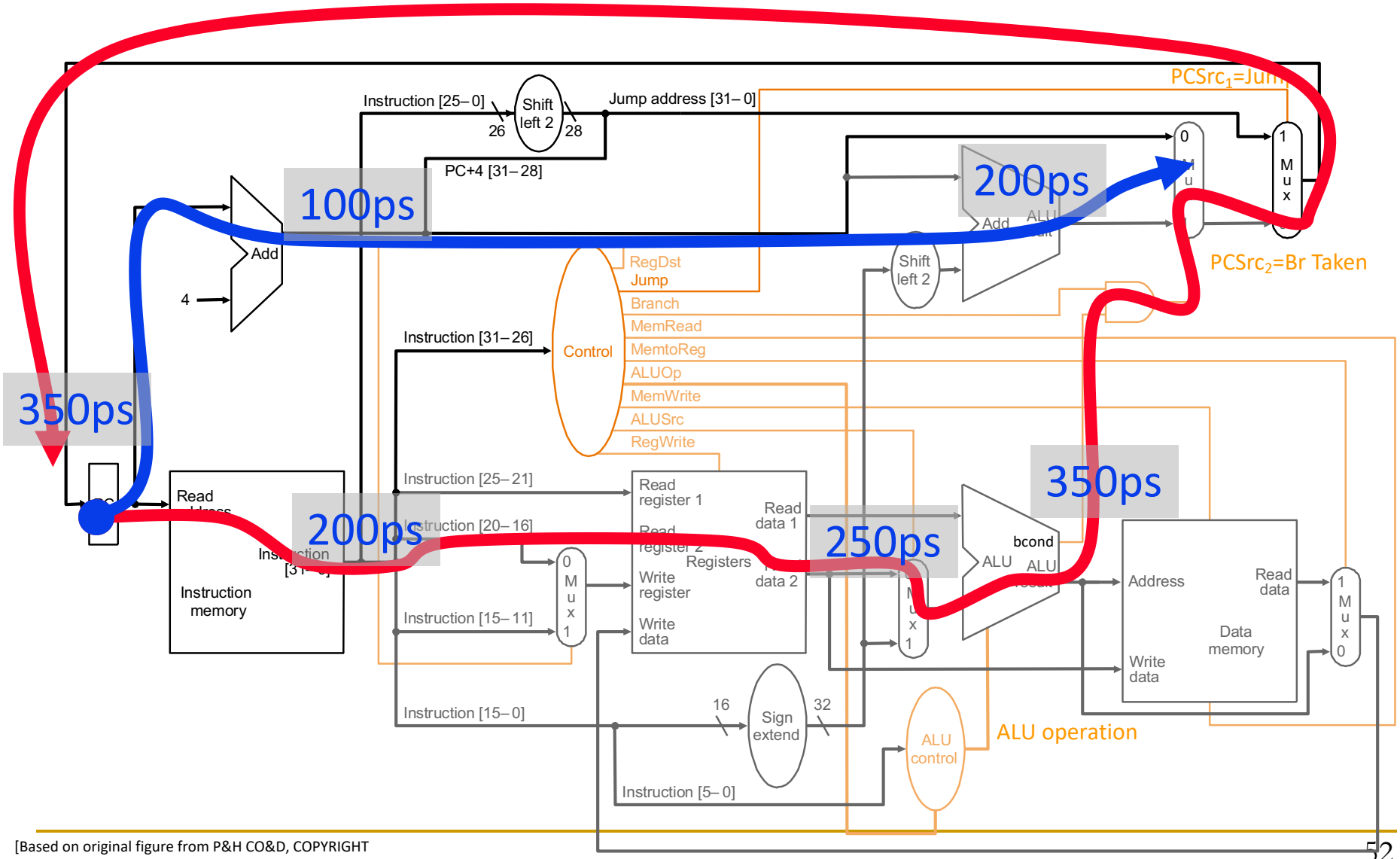
# LW



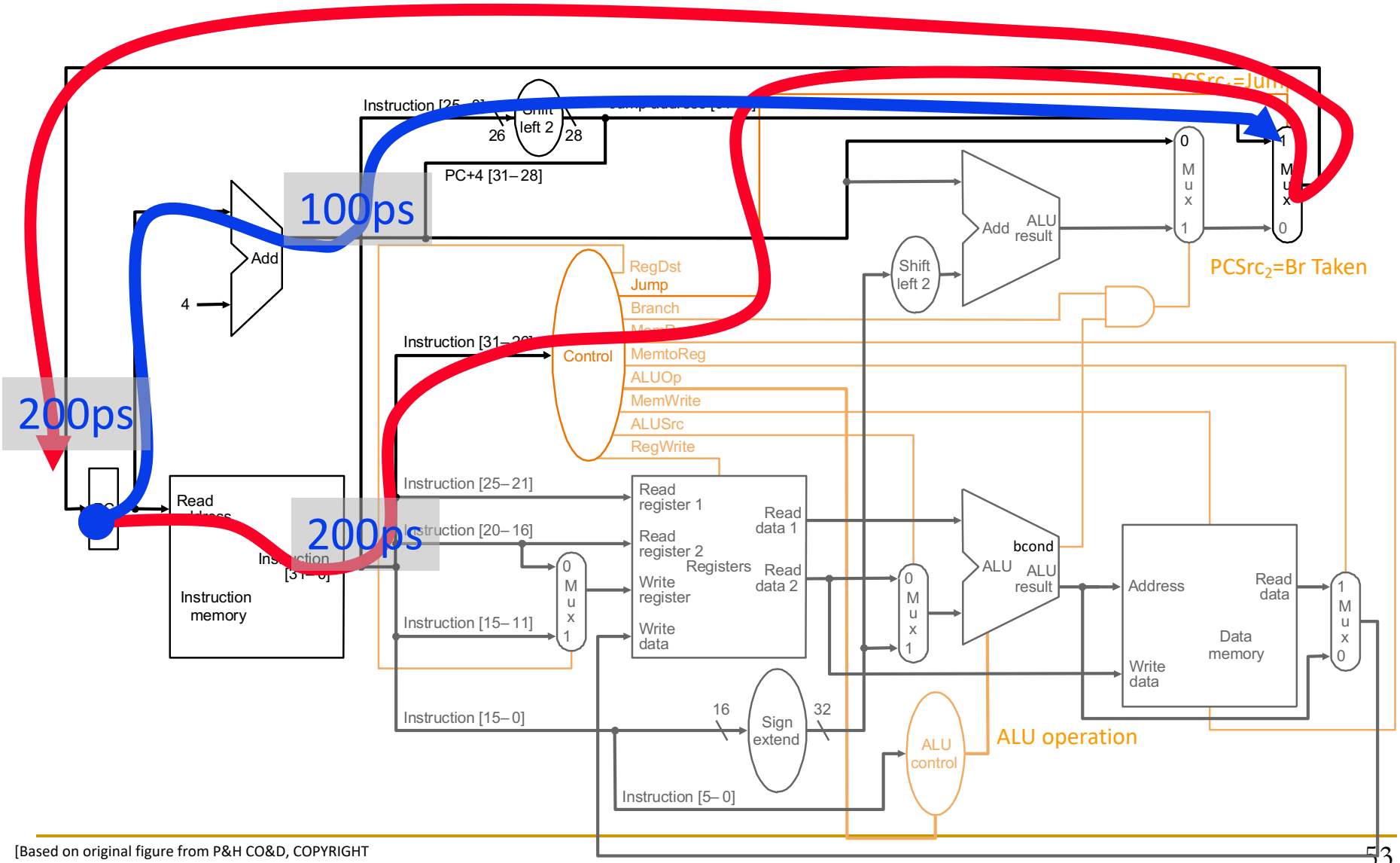
# SW



# Branch Taken



# Jump



# What About Control Logic?

- How does that affect the critical path?

- Food for thought for you:
  - Can control logic be on the critical path?
  - Historical example:
    - CDC 5600: control store access too long…

# What is the Slowest Instruction to Process?

- Memory is not magic
- What if memory *sometimes* takes 100ms to access?
- Does it make sense to have a simple register to register add or jump to take {100ms+all else to do a memory operation}?
- And, what if you need to access memory more than once to process an instruction?
  - Which instructions need this?
  - Do you provide multiple ports to memory?

# Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction
- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle
- Not necessarily the simplest way to implement an ISA
  - Single-cycle implementation of REP MOVS (x86) or INDEX (VAX)?
- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time

# (Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational logic delay
  - Break a path into multiple cycles if it takes too long

- Bread and butter (common case) design
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case

- Balanced design
  - Balance instruction/data flow through hardware components
  - Design to eliminate bottlenecks: balance the hardware for the work

# Single-Cycle Design vs. Design Principles

- Critical path design
- Bread and butter (common case) design
- Balanced design

*How does a single-cycle microarchitecture fare in light of these principles?*

# Aside: System Design Principles

- When designing computer systems/architectures, it is important to follow good principles

- Remember: "principled design" from our first lecture
  - Frank Lloyd Wright: "architecture [...] based upon principle, and not upon precedent"

# Aside: From Lecture 1

- "architecture [...] based upon principle, and not upon precedent"

# Aside: System Design Principles

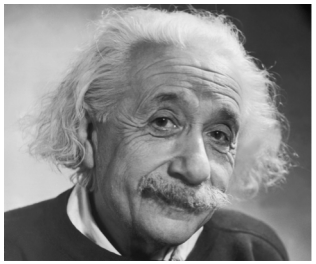
- We will continue to cover key principles in this course
- Here are some references where you can learn more

- Yale Patt, "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution," Proc. of IEEE, 2001. (Levels of transformation, design point, etc)
- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966. (Flynn's Bottleneck → Balanced design)
- Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS Conference, April 1967. (Amdahl's Law → Common-case design)
- Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.
  - http://research.microsoft.com/pubs/68221/acrobat.pdf

# A Key System Design Principle

- Keep it simple

- "Everything should be made as simple as possible, but no simpler."
  - Albert Einstein



- And, keep it low cost: "An engineer is a person who can do for a dime what any fool can do for a dollar."

- For more, see:
  - Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.
  - http://research.microsoft.com/pubs/68221/acrobat.pdf

# Multi-Cycle Microarchitectures

# Multi-Cycle Microarchitectures

- Goal: Let each instruction take (close to) only as much time it really needs

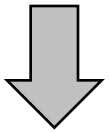
- Idea
  - Determine clock cycle time independently of instruction processing time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different

# Remember: The "Process instruction" Step

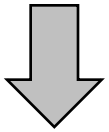
- ISA specifies abstractly what AS' should be, given an instruction and AS
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no "intermediate states" between AS and AS' during instruction execution
    - One state transition per instruction
- Microarchitecture implements how AS is transformed to AS'
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
    - Choice 1: AS → AS' (transform AS to AS' in a single clock cycle)
    - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

# Multi-Cycle Microarchitecture

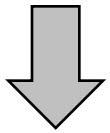
AS = Architectural (programmer visible) state
at the beginning of an instruction



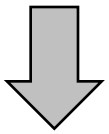
Step 1: Process part of instruction in one clock cycle



Step 2: Process part of instruction in the next clock cycle



...



AS' = Architectural (programmer visible) state
at the end of a clock cycle

# Benefits of Multi-Cycle Design

- Critical path design
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction
- Bread and butter (common case) design
  - Can optimize the number of states it takes to execute "important" instructions that make up much of the execution time
- Balanced design
  - No need to provide more capability or resources than really needed
    - An instruction that needs resource X multiple times does not require multiple X's to be implemented
    - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction

# Downsides of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for registers
  - Register setup/hold overhead paid multiple times for an instruction

# Remember: Performance Analysis

- Execution time of an instruction
  - {CPI} x {clock cycle time}
- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**
- Single cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

**Not easy to optimize design**

**We have two degrees of freedom to optimize independently**

# A Multi-Cycle Microarchitecture
## *A Closer Look*

# How Do We Implement This?

- Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.

THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

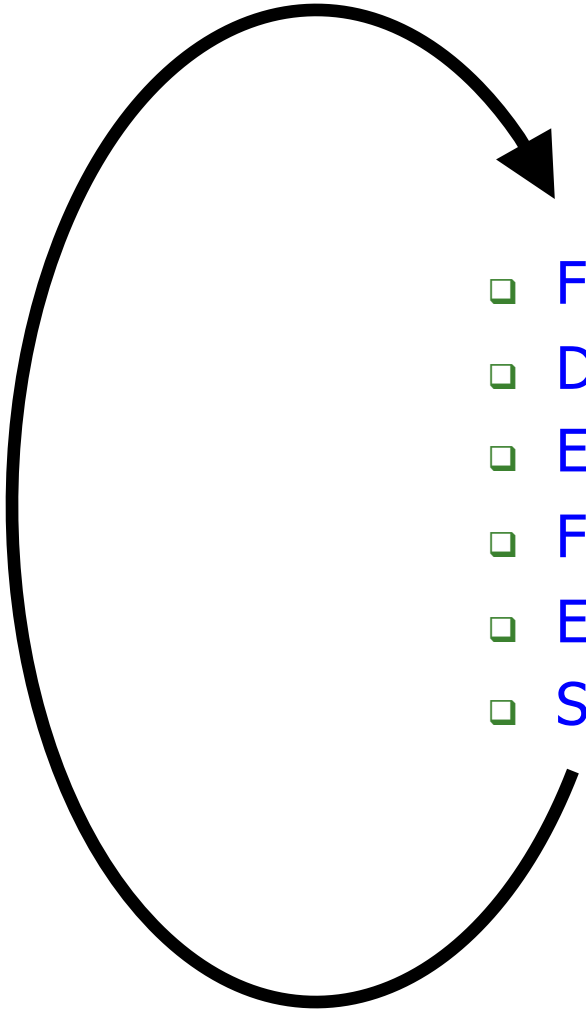
By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines

# Multi-Cycle uArch

- Key Idea for Realization
  - One can implement the "process instruction" step as a finite state machine that sequences between states and eventually returns back to the "fetch instruction" state
  - A state is defined by the control signals asserted in it
  - Control signals for the next state are determined in current state

# The Instruction Processing Cycle



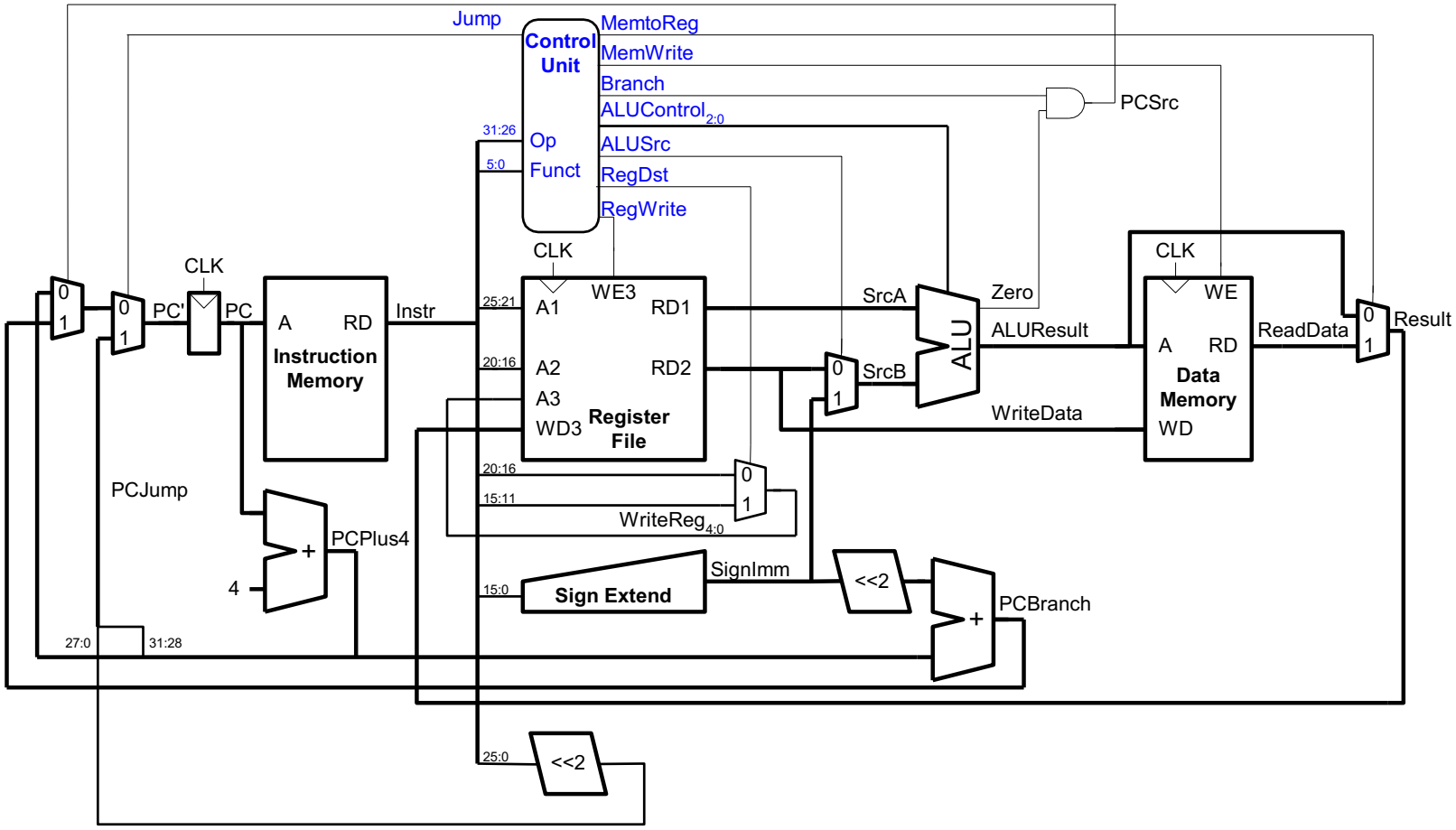
- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

# A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into "states"
  - A stage in the instruction processing cycle can take multiple states
- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state
- The behavior of the entire processor is specified fully by a *finite state machine*
- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle

# One Example Multi-Cycle Microarchitecture

# Remember: Single-Cycle MIPS Processor



# Multi-cycle MIPS Processor

- **Single-cycle microarchitecture:**
  - cycle time limited by longest instruction (`lw`) → low clock frequency
  - three adders/ALUs and two memories → high hardware cost
- **Multi-cycle microarchitecture:**
  + higher clock frequency
  + simpler instructions run faster
  + reuse expensive hardware across multiple cycles
  - sequencing overhead paid many times
  - hardware overhead for storing intermediate results
- **Same design steps: datapath & control**

# What Do We Want To Optimize

- **Single Cycle Architecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)

# What Do We Want To Optimize

- **Single Cycle Architecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)
- **Single Cycle Architecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)

# What Do We Want To Optimize

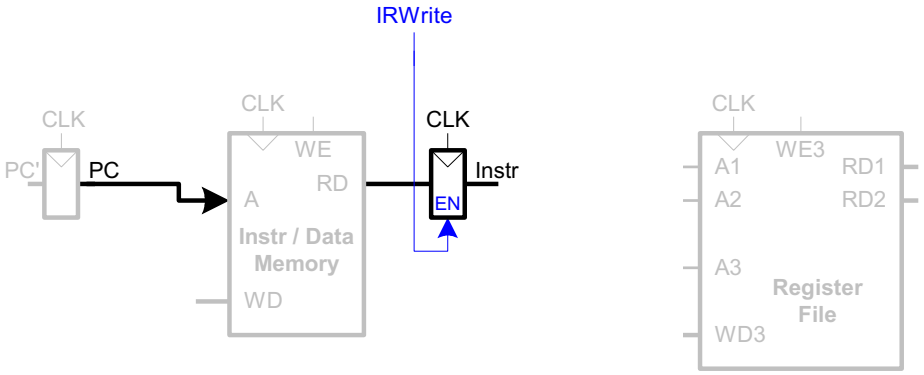
- **Single Cycle Architecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)
- **Single Cycle Architecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)
- **In Single Cycle Architecture all instructions take one cycle**
  - The most complex operation slows down everything!
  - Divide all instructions into multiple steps
  - Simpler instructions can take fewer cycles (average case may be faster)

# Consider the lw instruction

- **For an instruction such as: `lw $t0, 0x20($t1)`**
- **We need to:**
  - Read the instruction from memory
  - Then read **`$t1`** from register array
  - Add the immediate value (**`0x20`**) to calculate the memory address
  - Read the content of this address
  - Write to the register **`$t0`** this content

# Multi-cycle Datapath: instruction fetch

- **First consider executing lw**
  - STEP 1: Fetch instruction

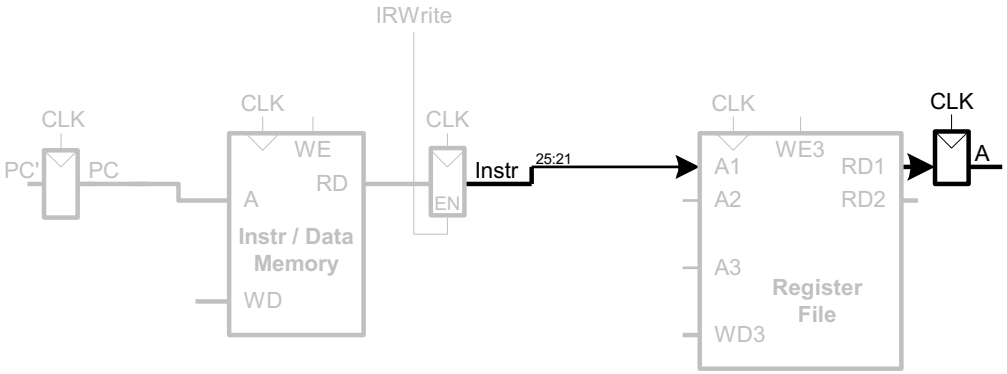



`read from the memory location [rs]+imm to location [rt]`

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

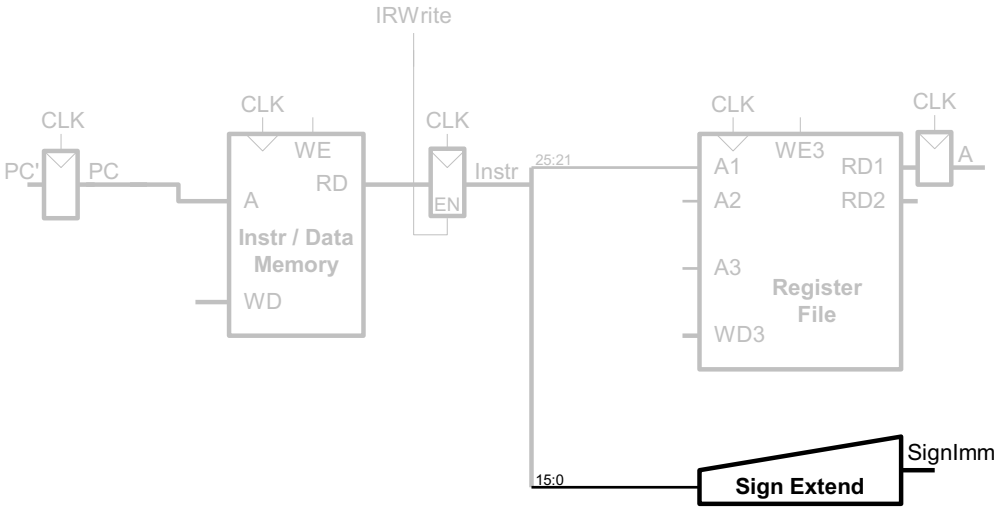
# Multi-cycle Datapath: `lw` register read




**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

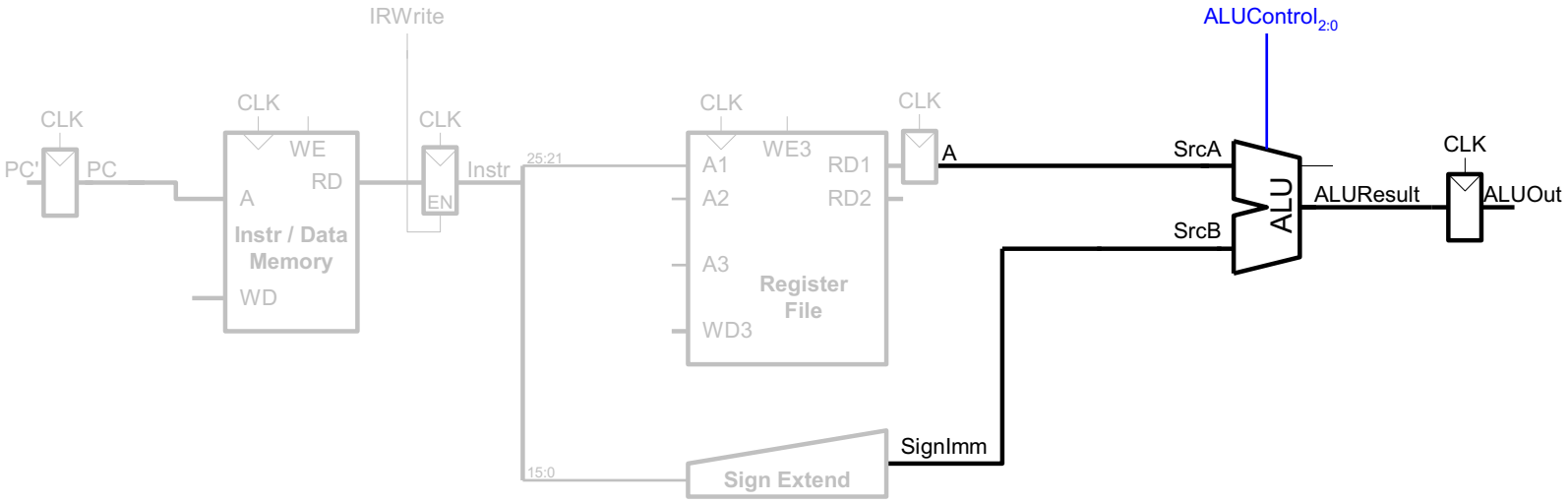
# Multi-cycle Datapath: lw immediate




## I-Type

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

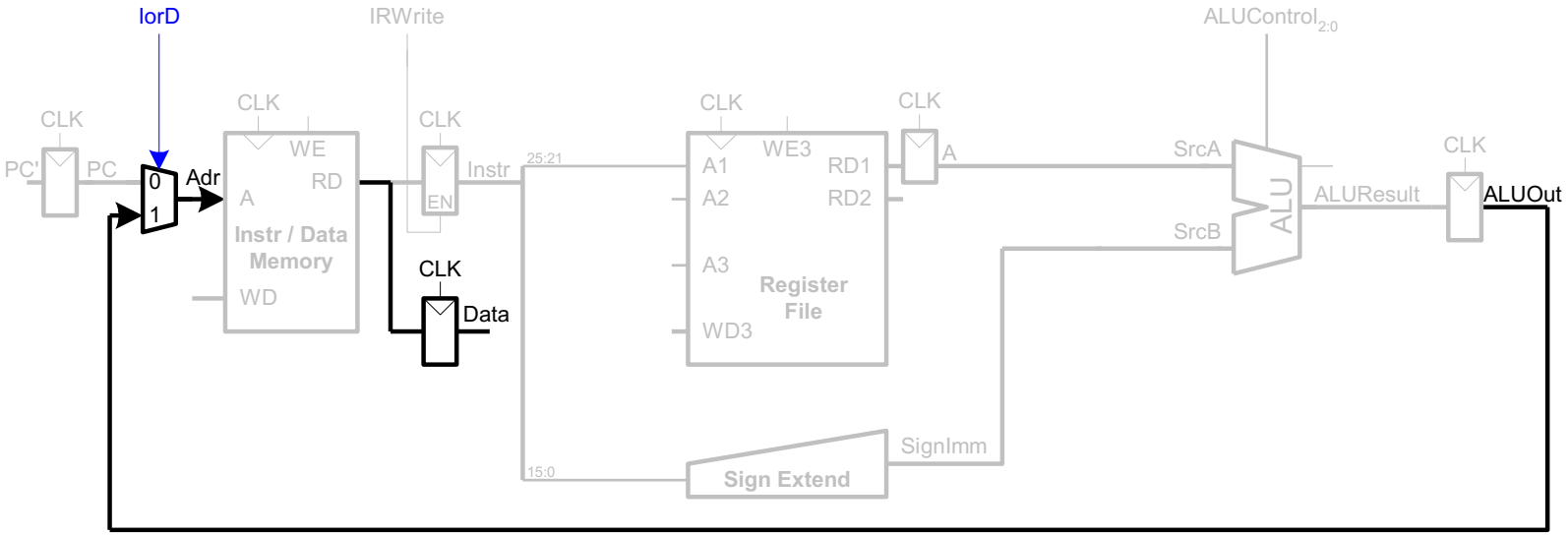
# Multi-cycle Datapath: lw address




## I-Type

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

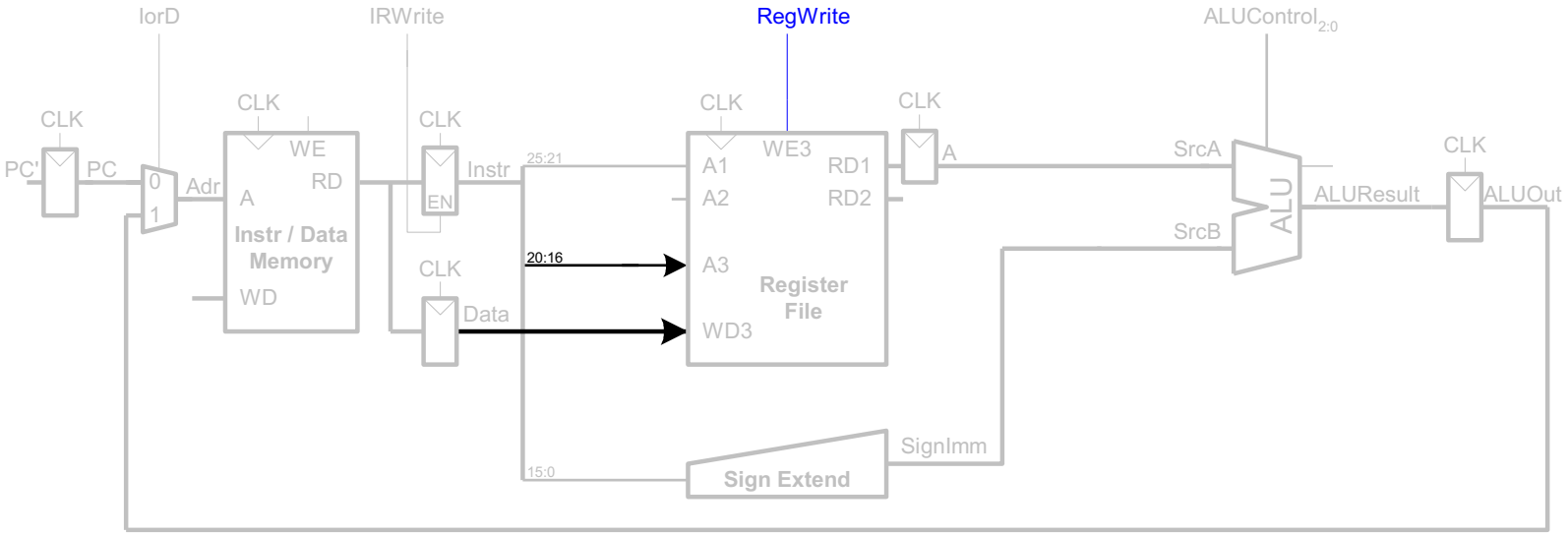
# Multi-cycle Datapath: lw memory read




## I-Type

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

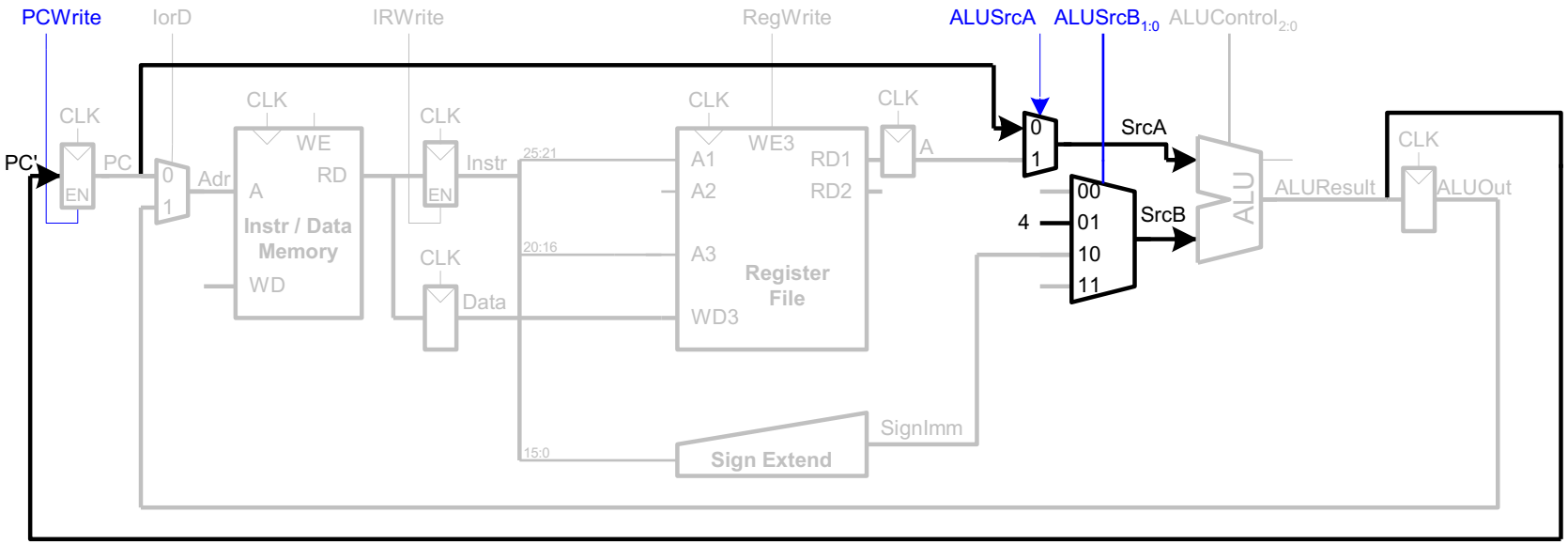
# Multi-cycle Datapath: `lw` write register




## I-Type

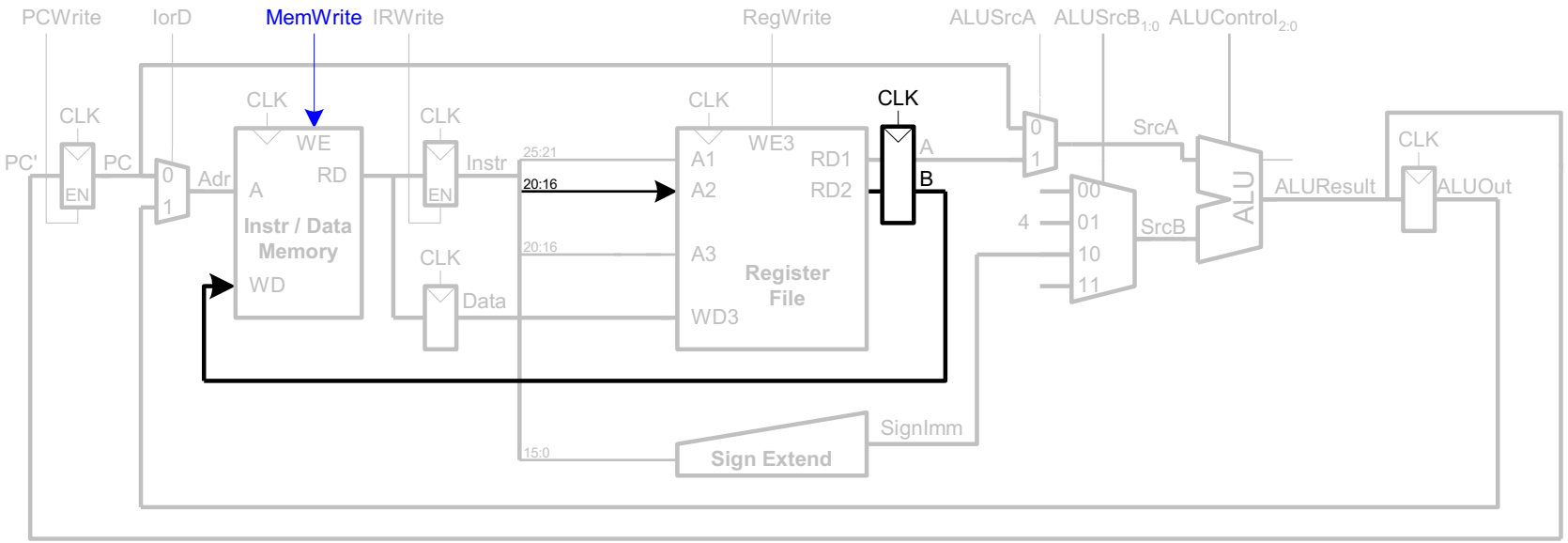
| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Multi-cycle Datapath: increment PC



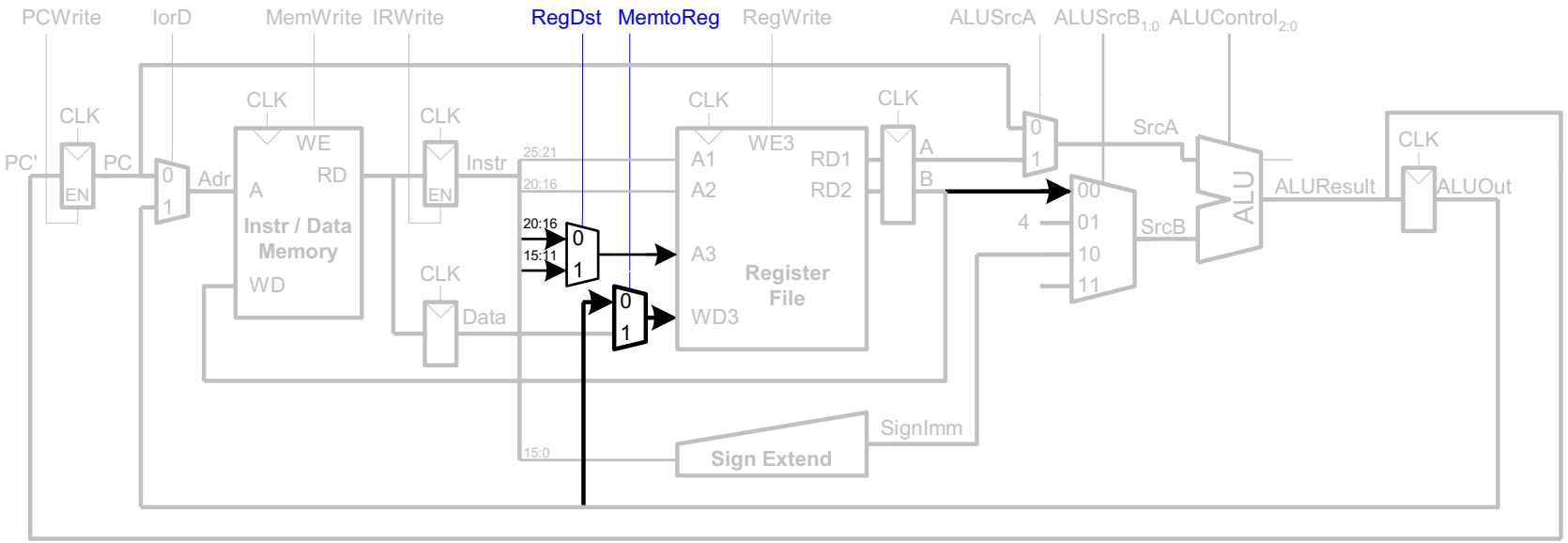
# Multi-cycle Datapath: sw

- Write data in rt to memory



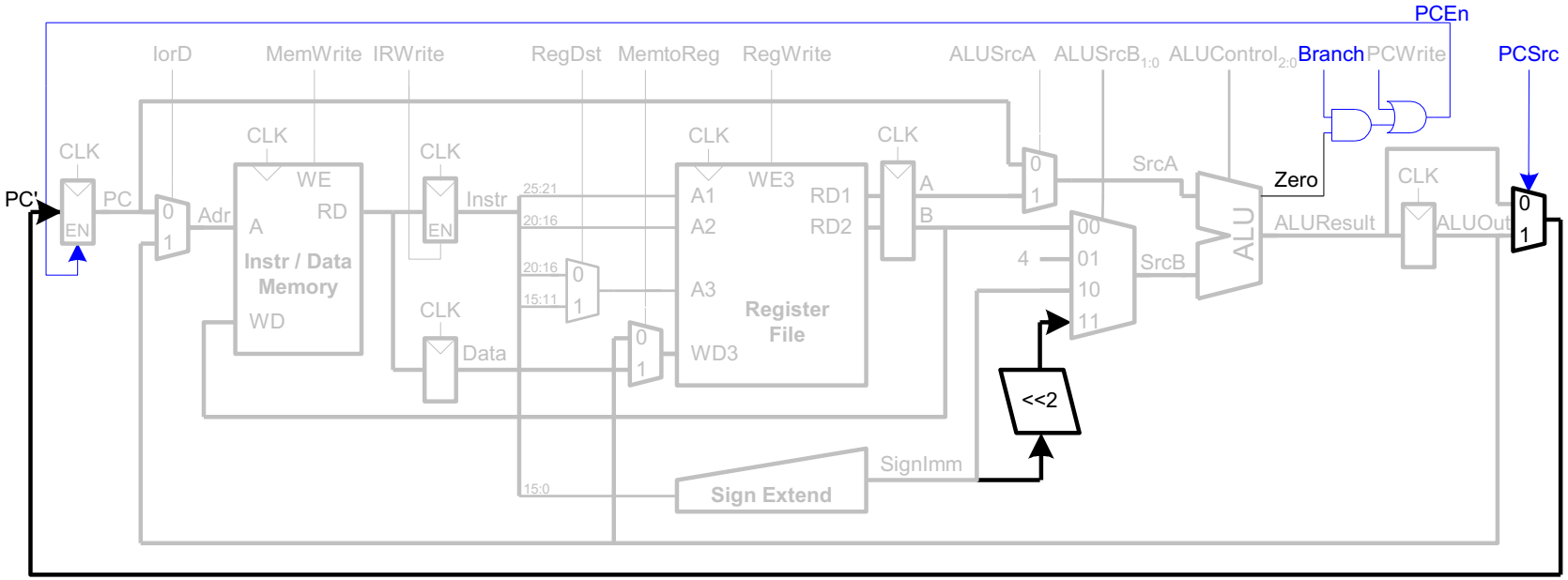
# Multi-cycle Datapath: R-type Instructions

- **Read from rs and rt**
  - Write ALUResult to register file
  - Write to rd (instead of rt)

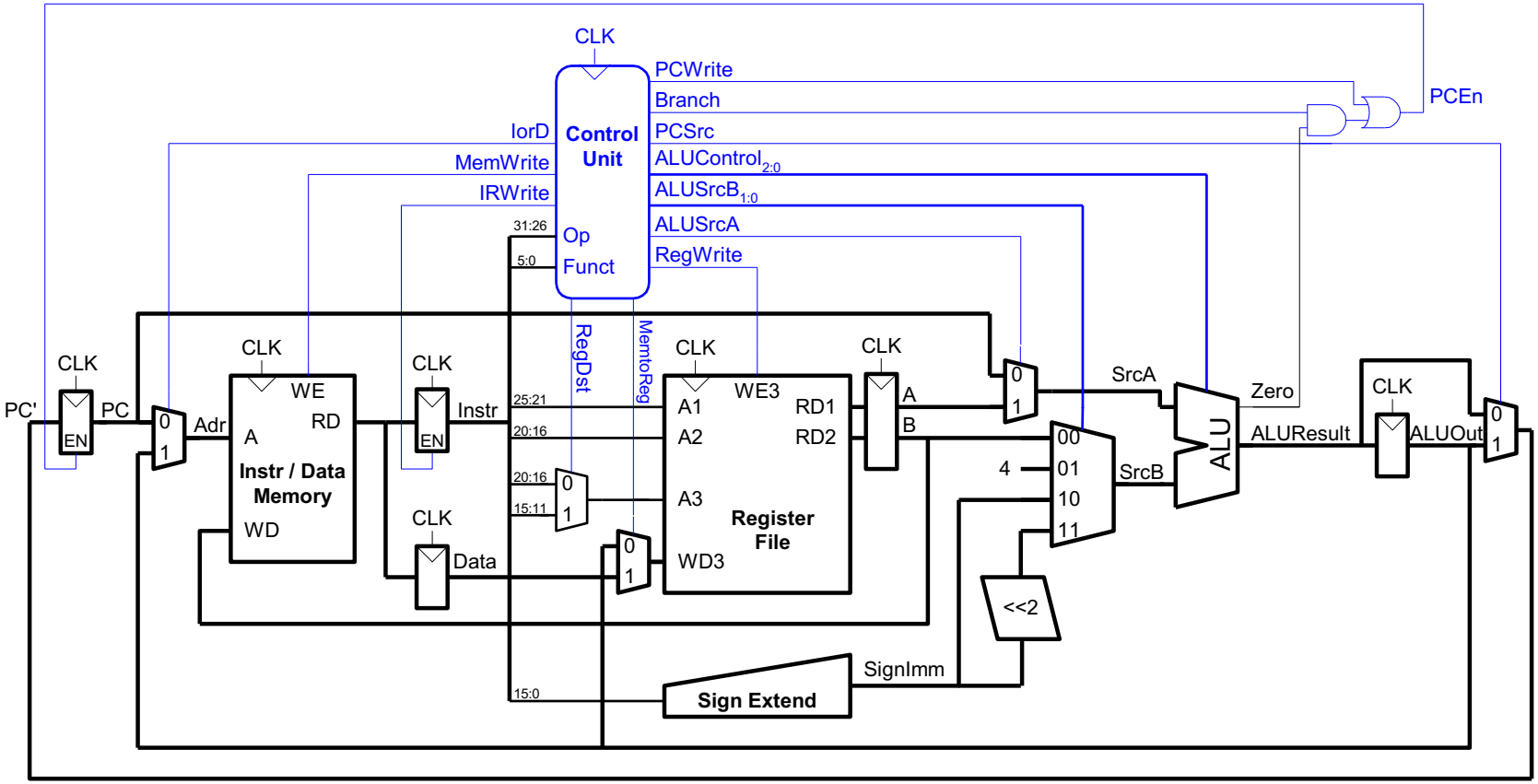


# Multi-cycle Datapath: beq

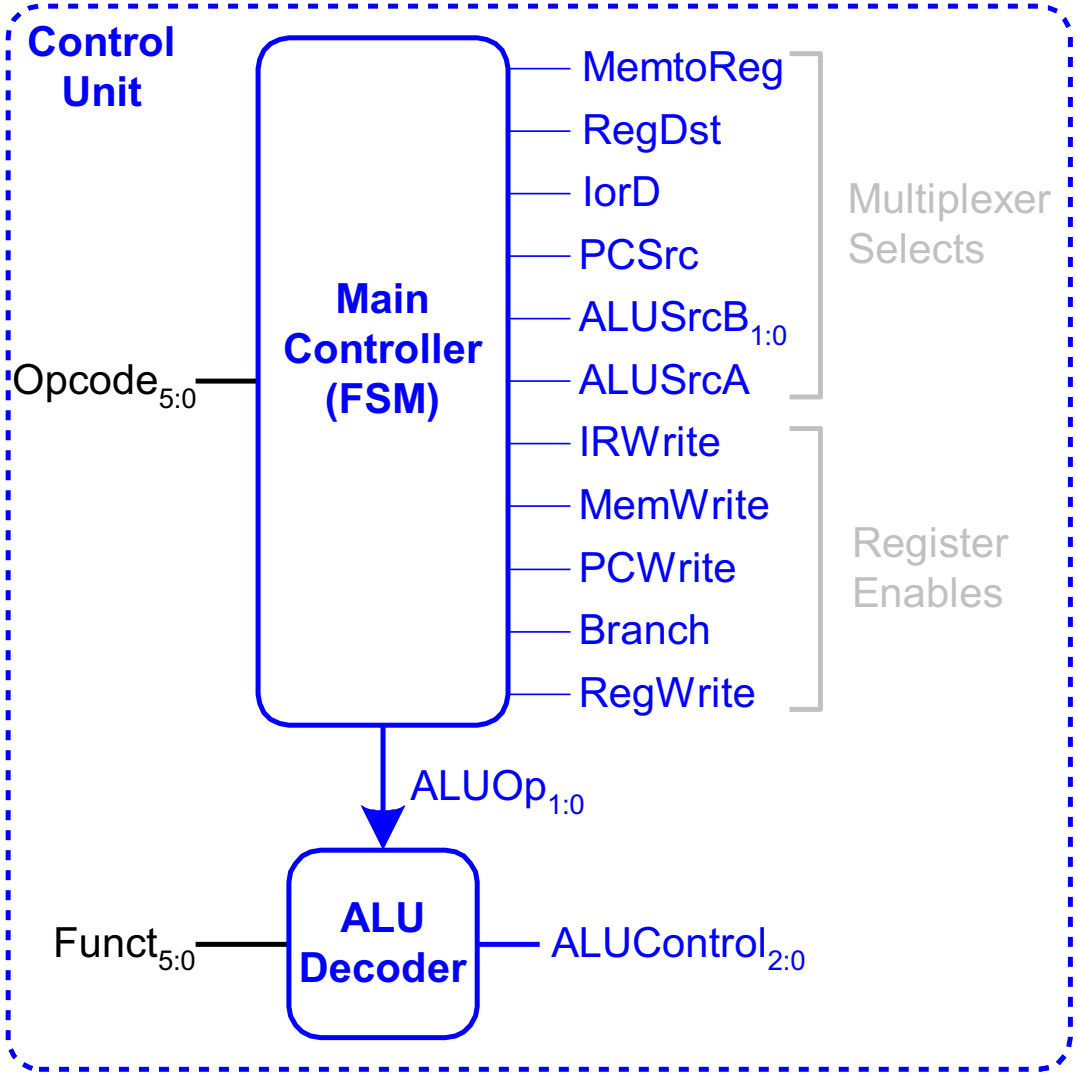
- **Determine whether values in rs and rt are equal**
  - Calculate branch target address:
    ***BTA*** = (sign-extended immediate << 2) + (PC+4)



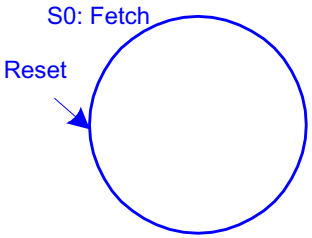
# Complete Multi-cycle Processor

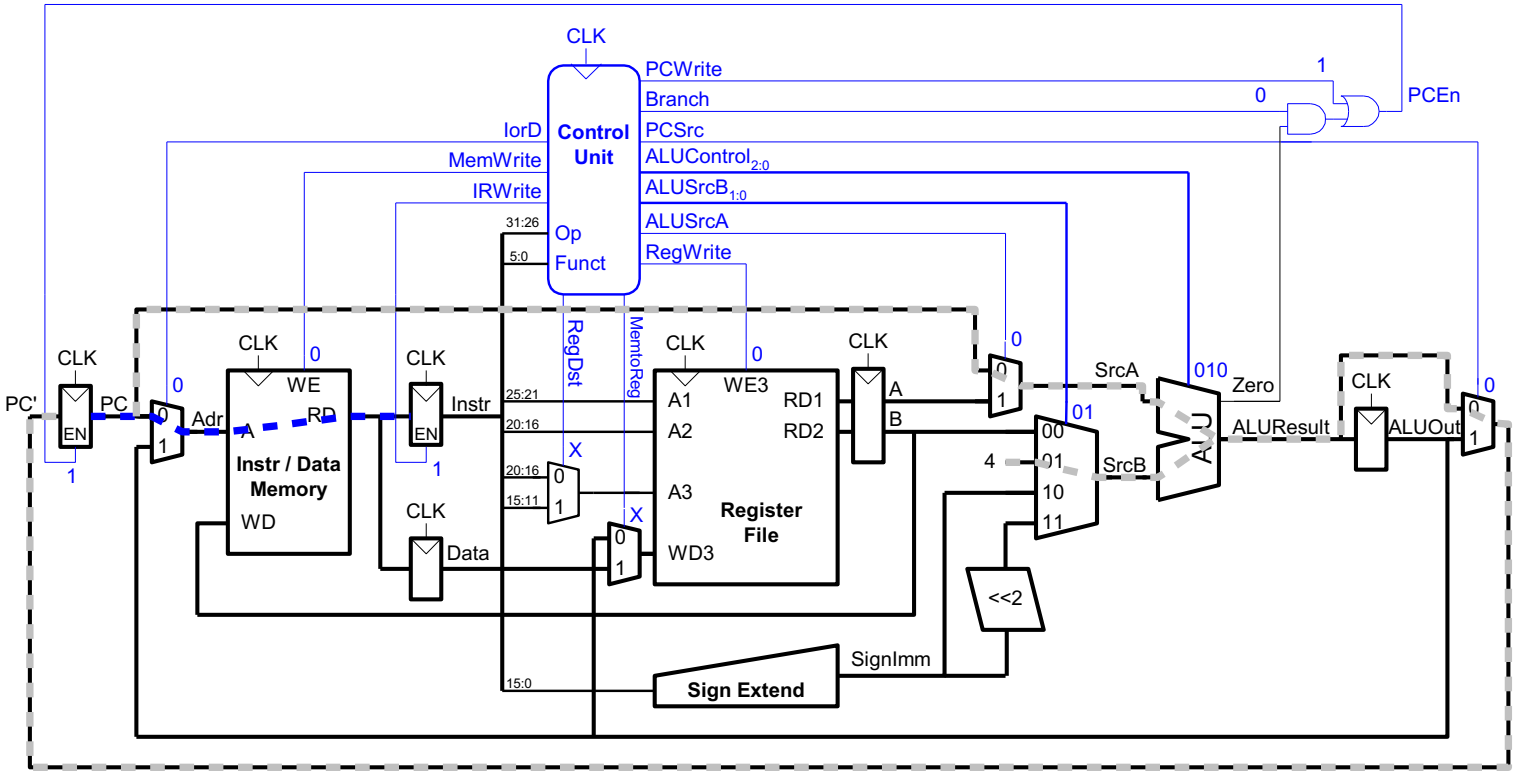


# Control Unit

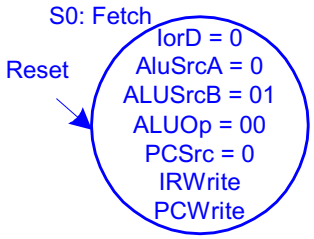


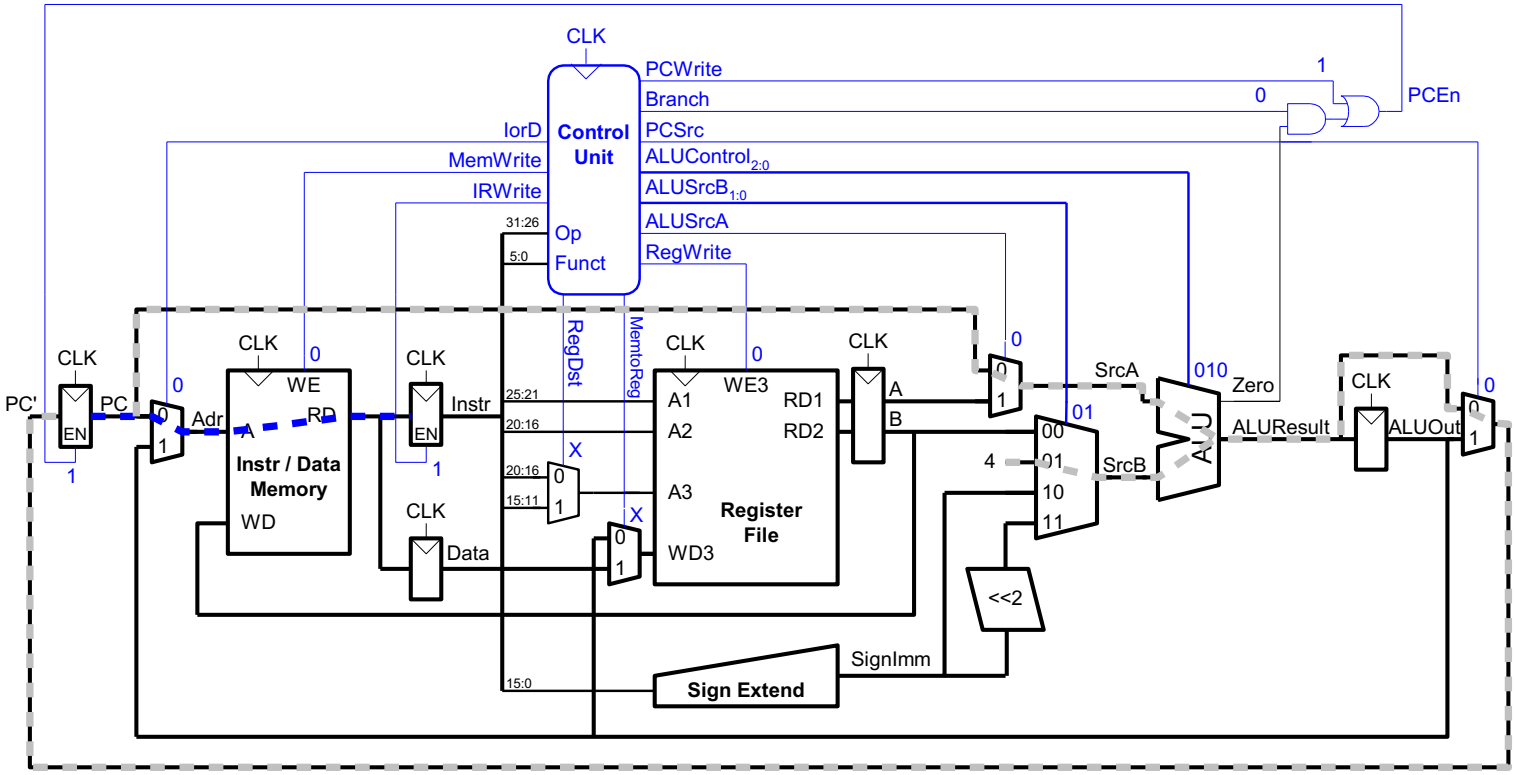
# Main Controller FSM: Fetch



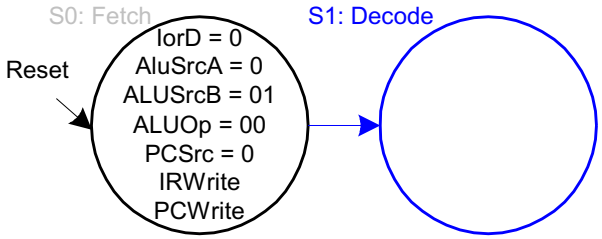



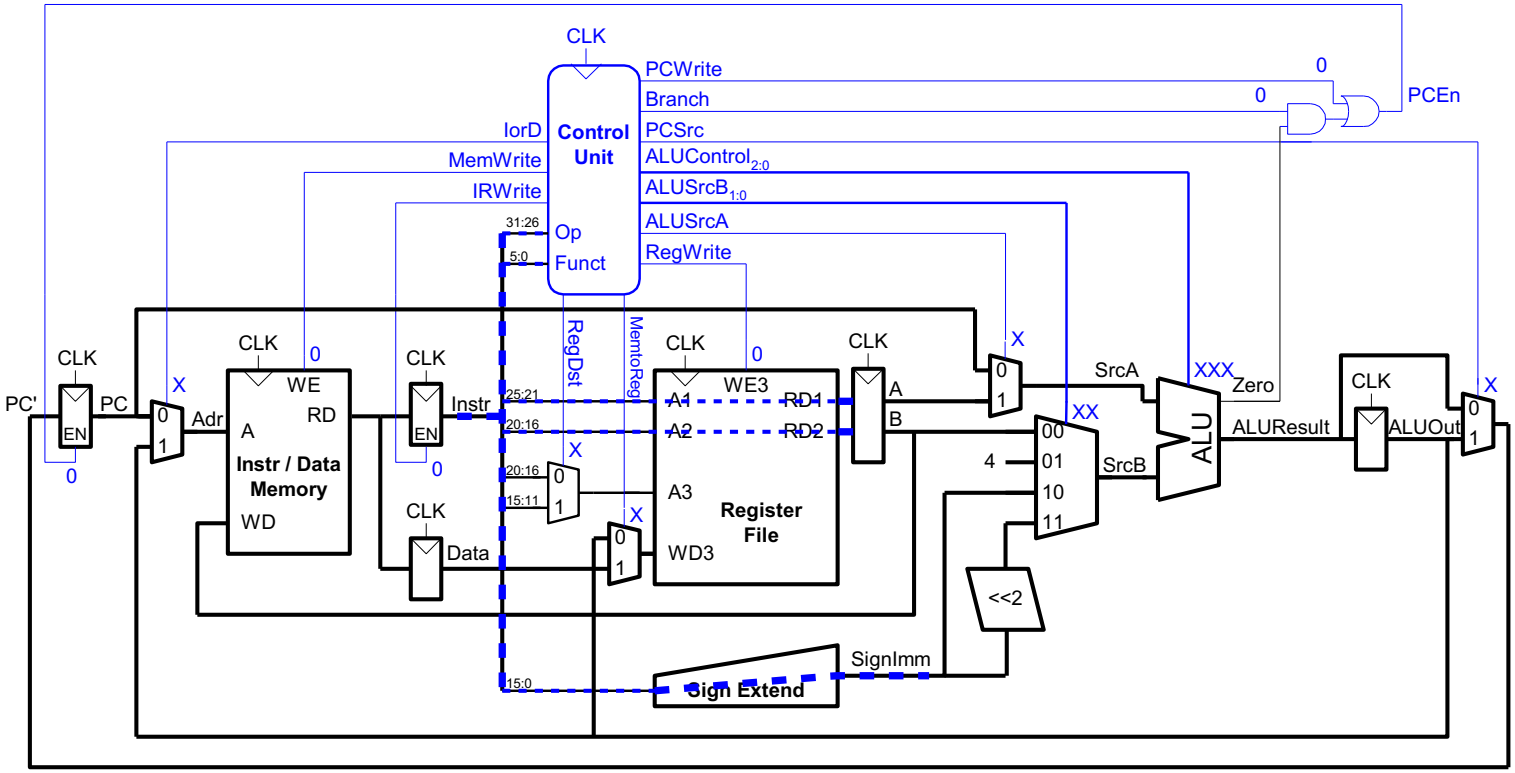
# Main Controller FSM: Fetch



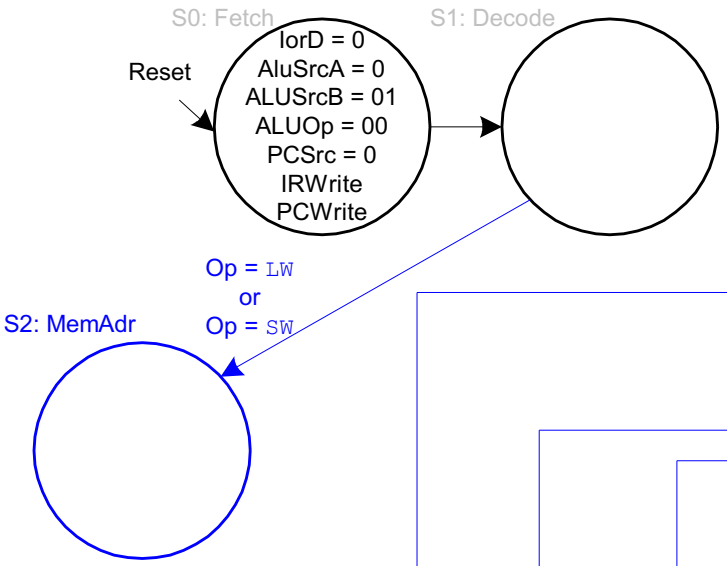



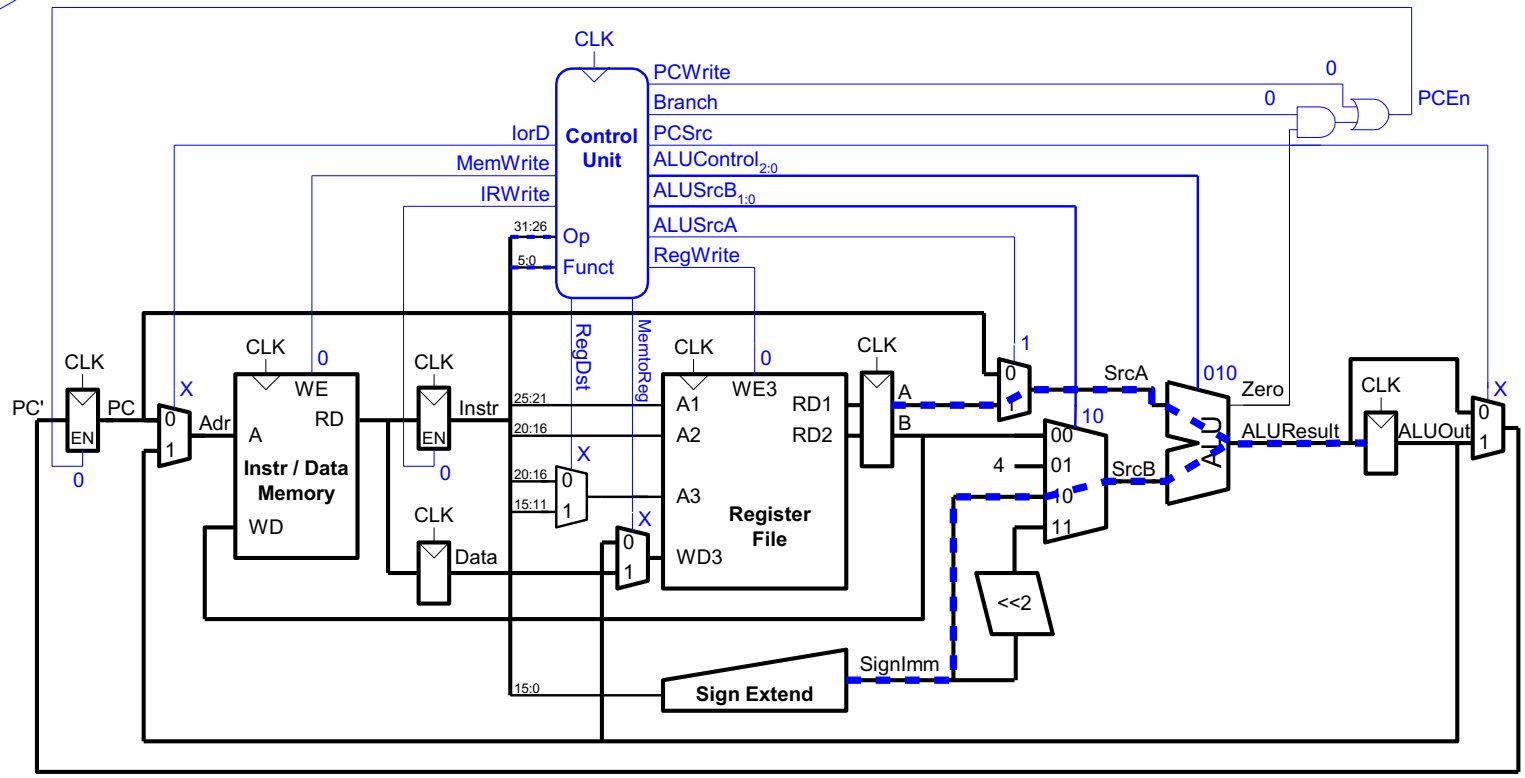
# Main Controller FSM: Decode



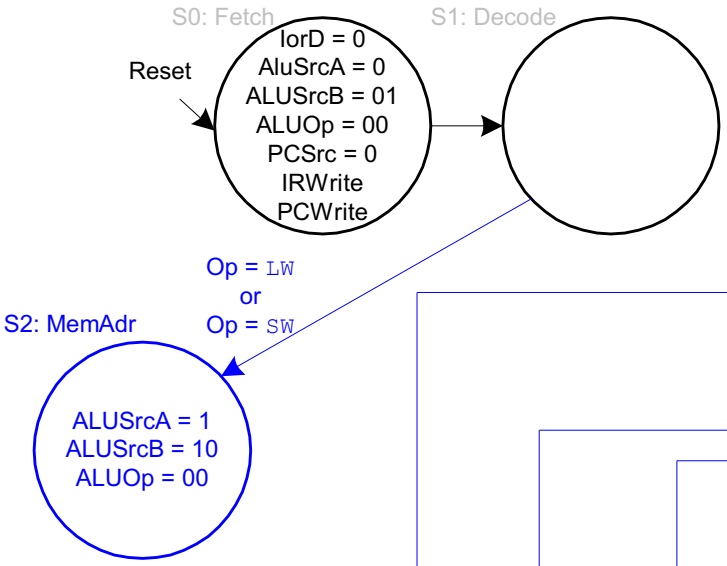



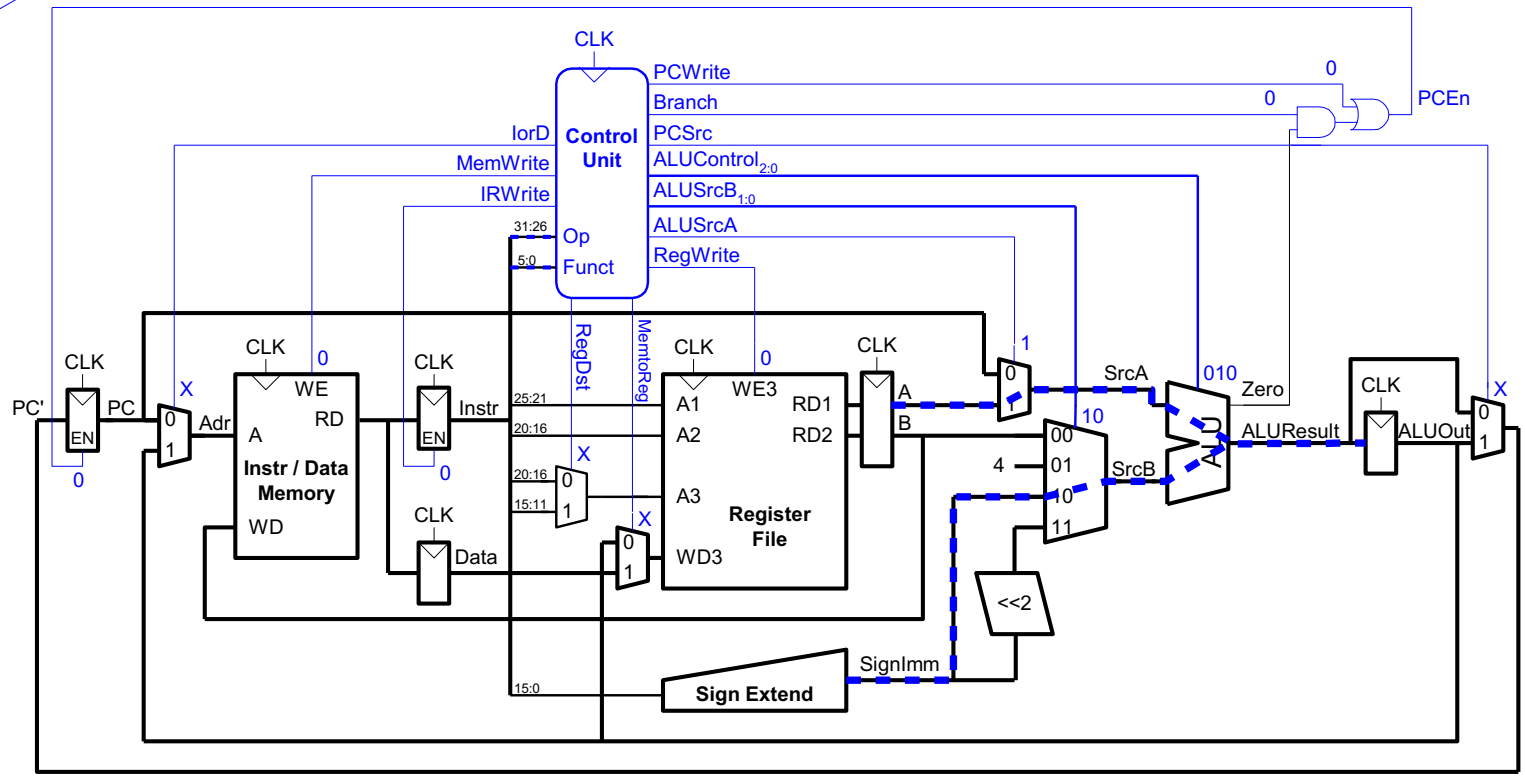
# Main Controller FSM: Address Calculation



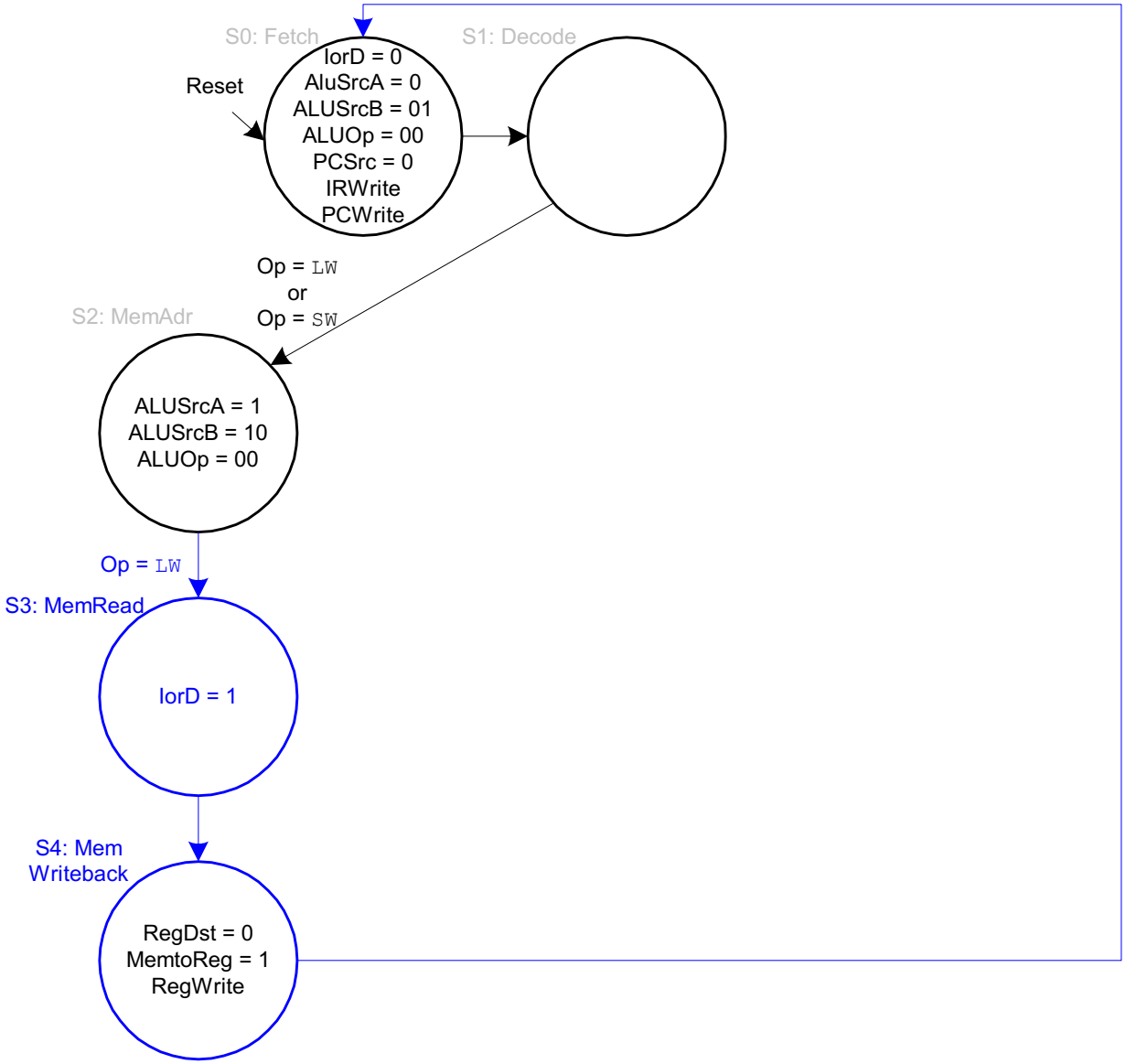



# Main Controller FSM: Address Calculation

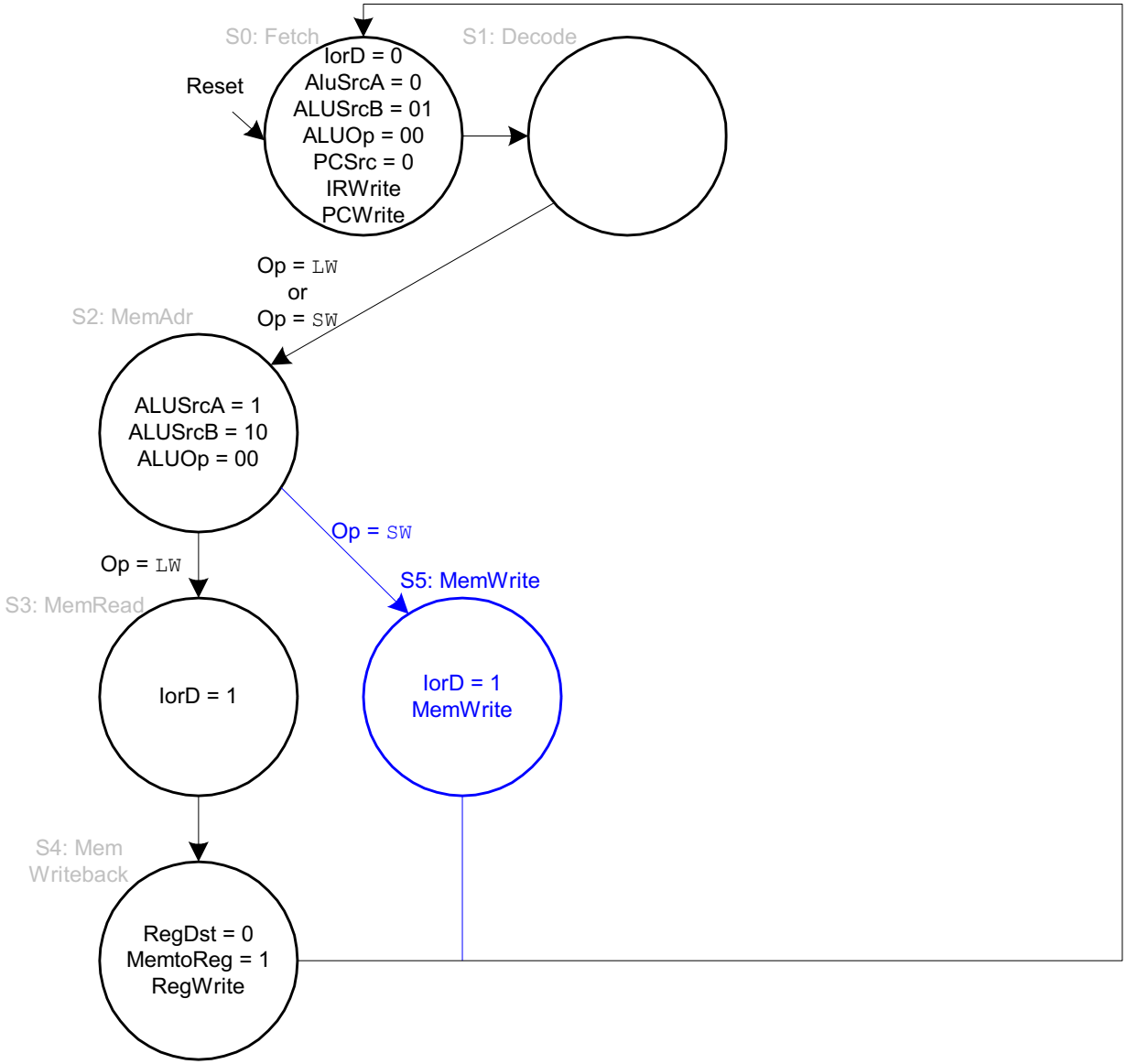


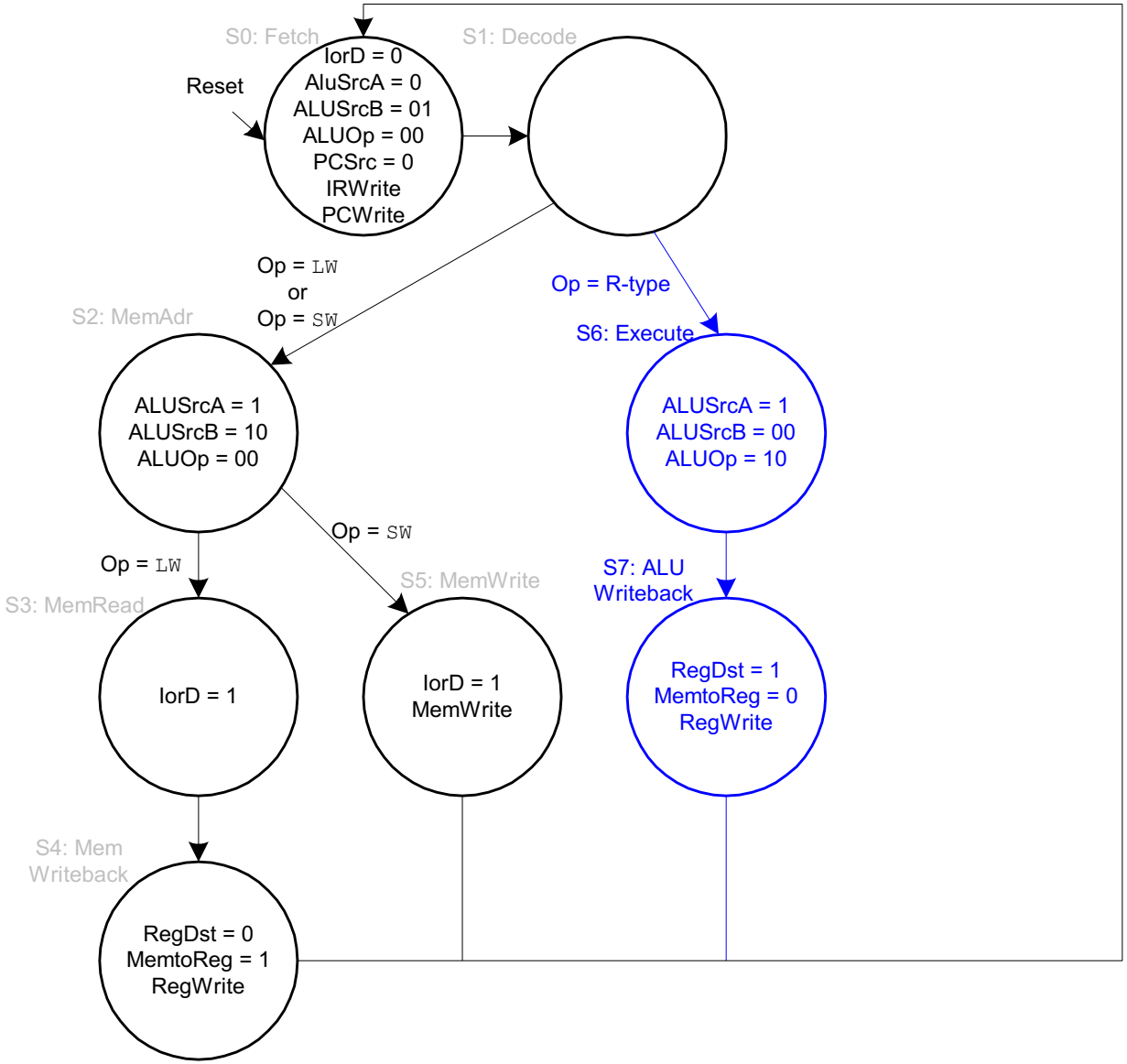
# Main Controller FSM: lw



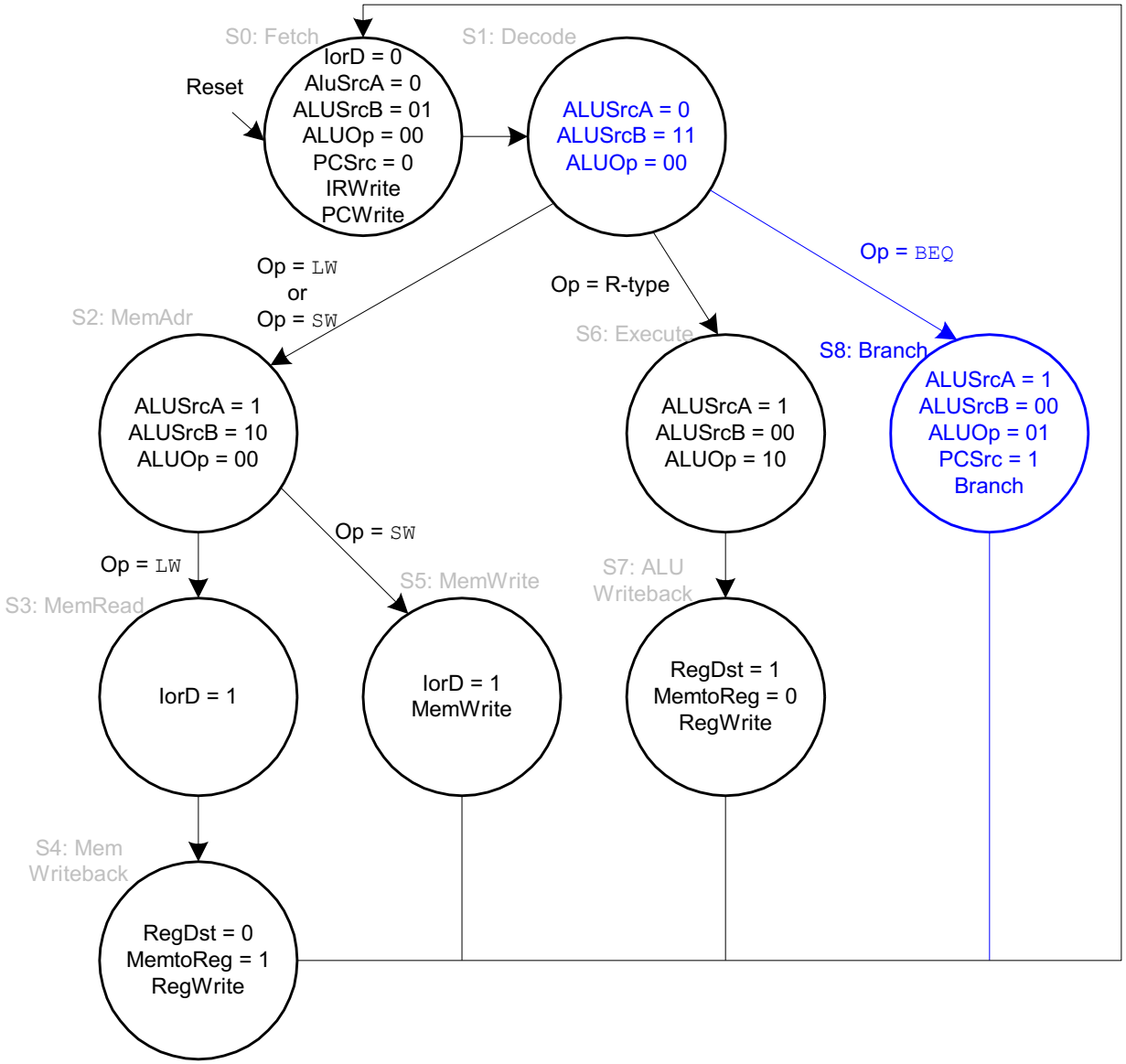
# Main Controller FSM: sw



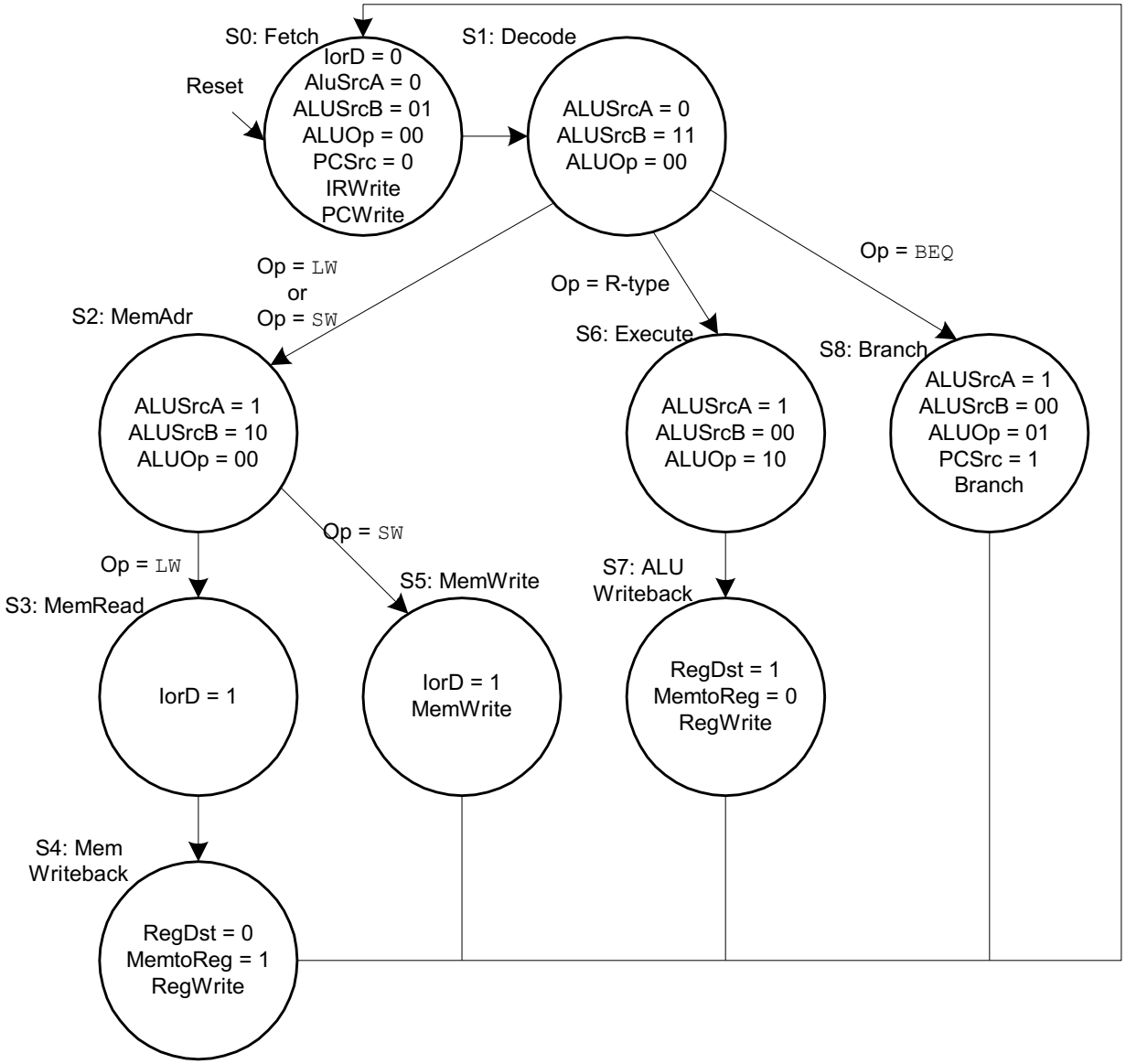
# Main Controller FSM: R-Type



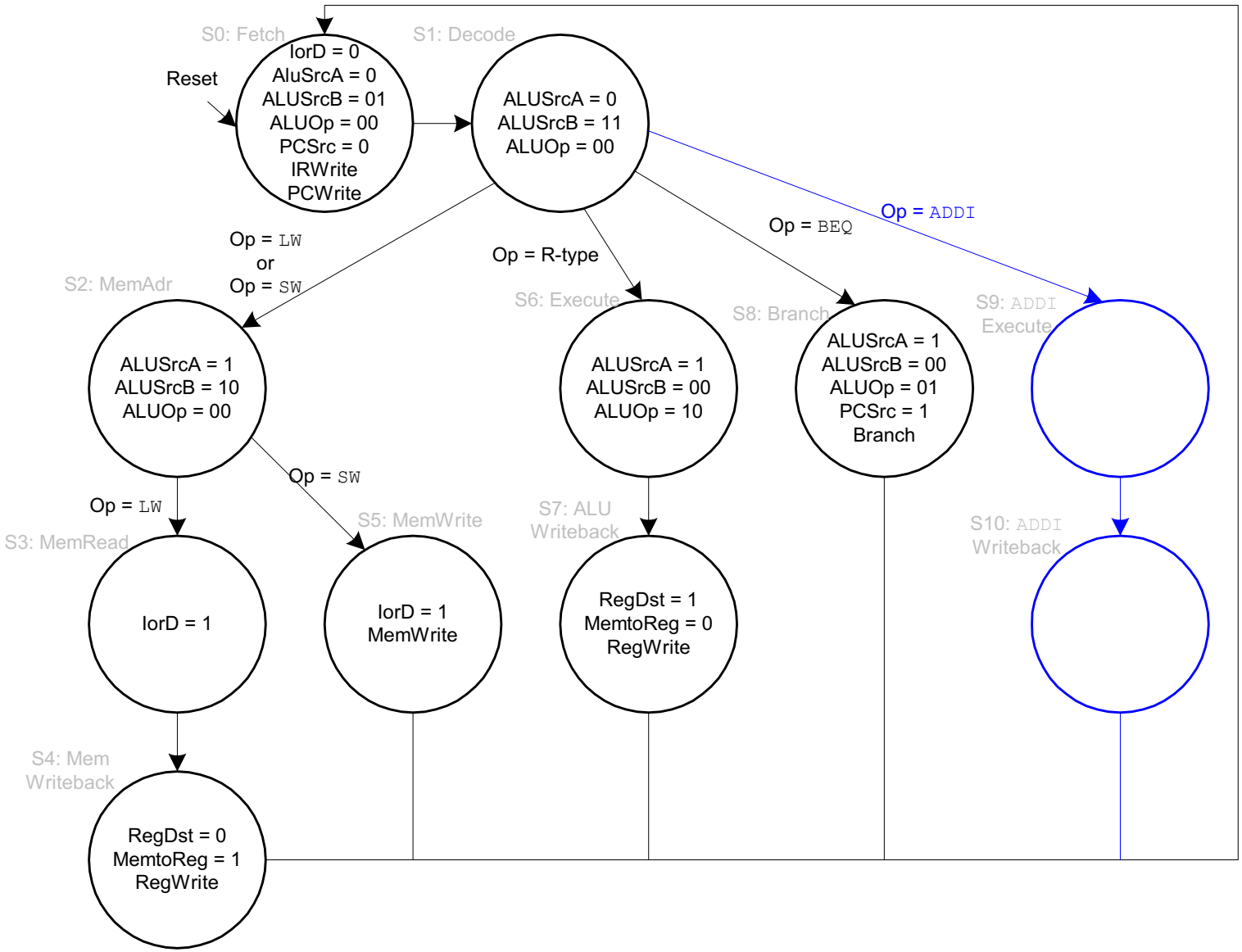
# Main Controller FSM: beq



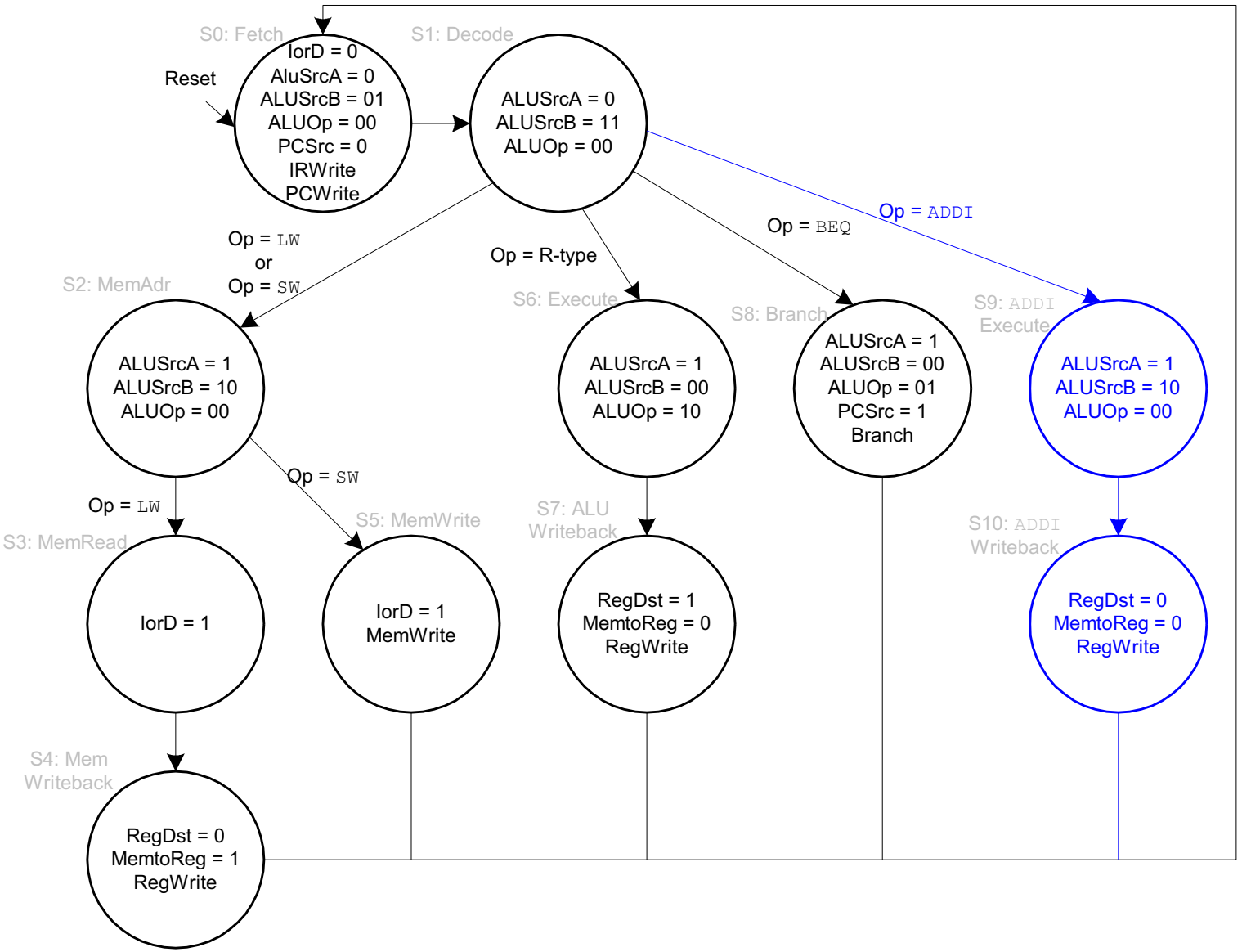
# Complete Multi-cycle Controller FSM



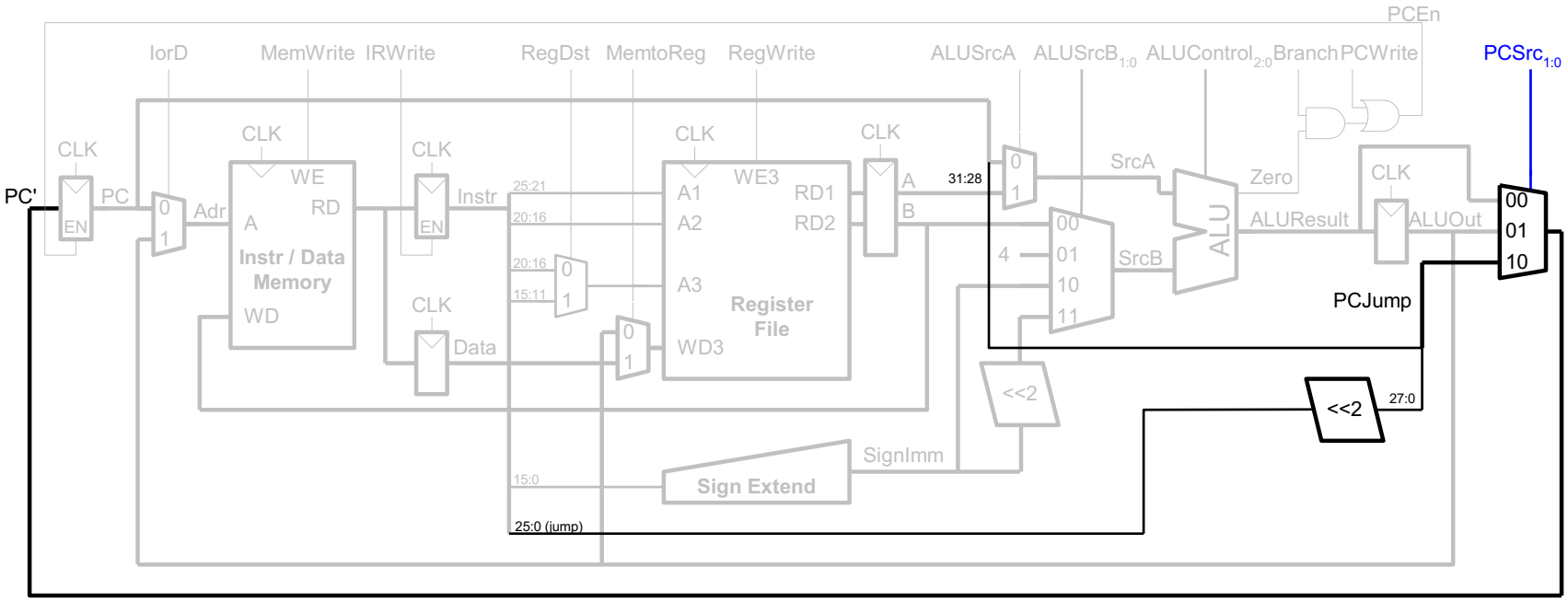
# Main Controller FSM: addi



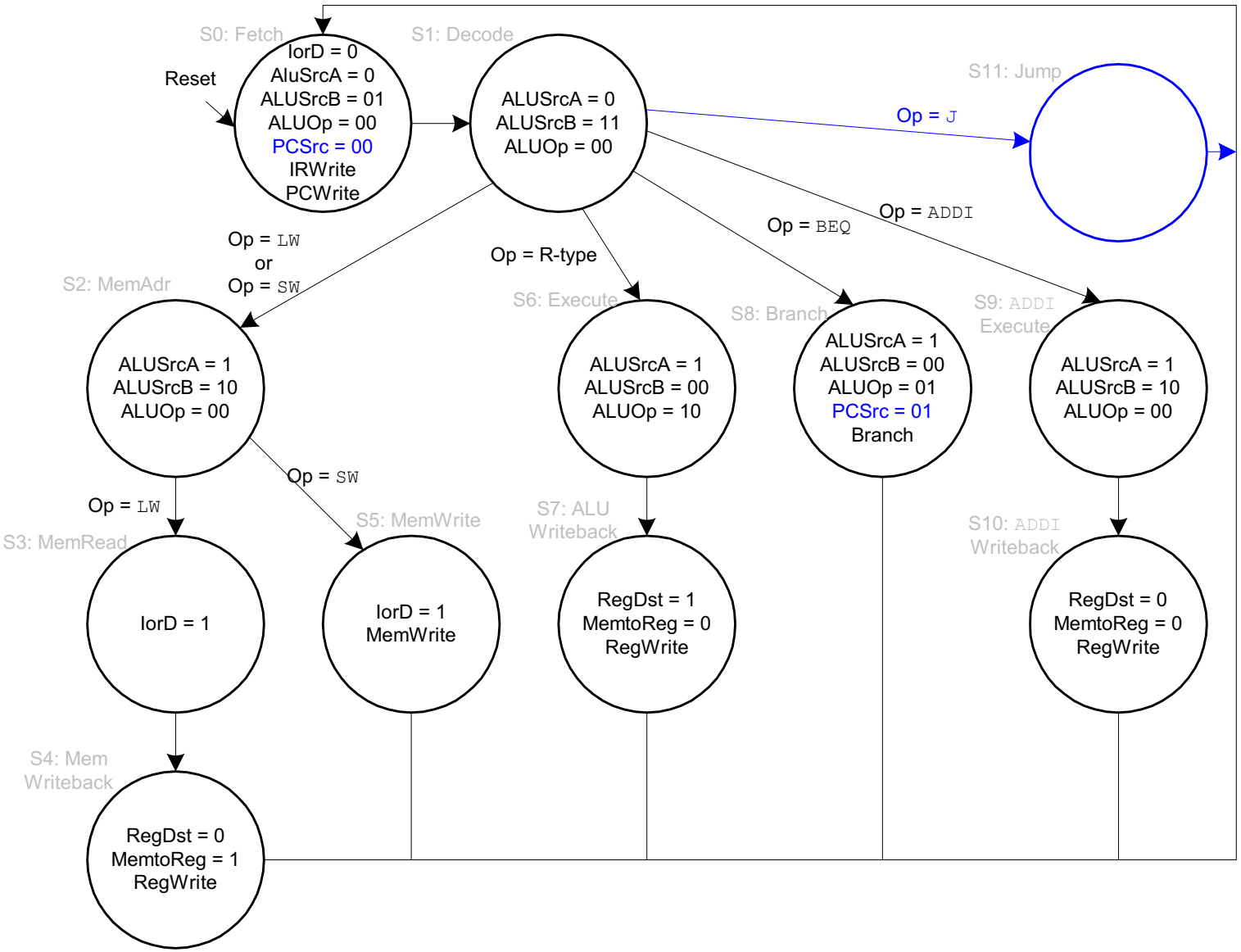
# Main Controller FSM: addi



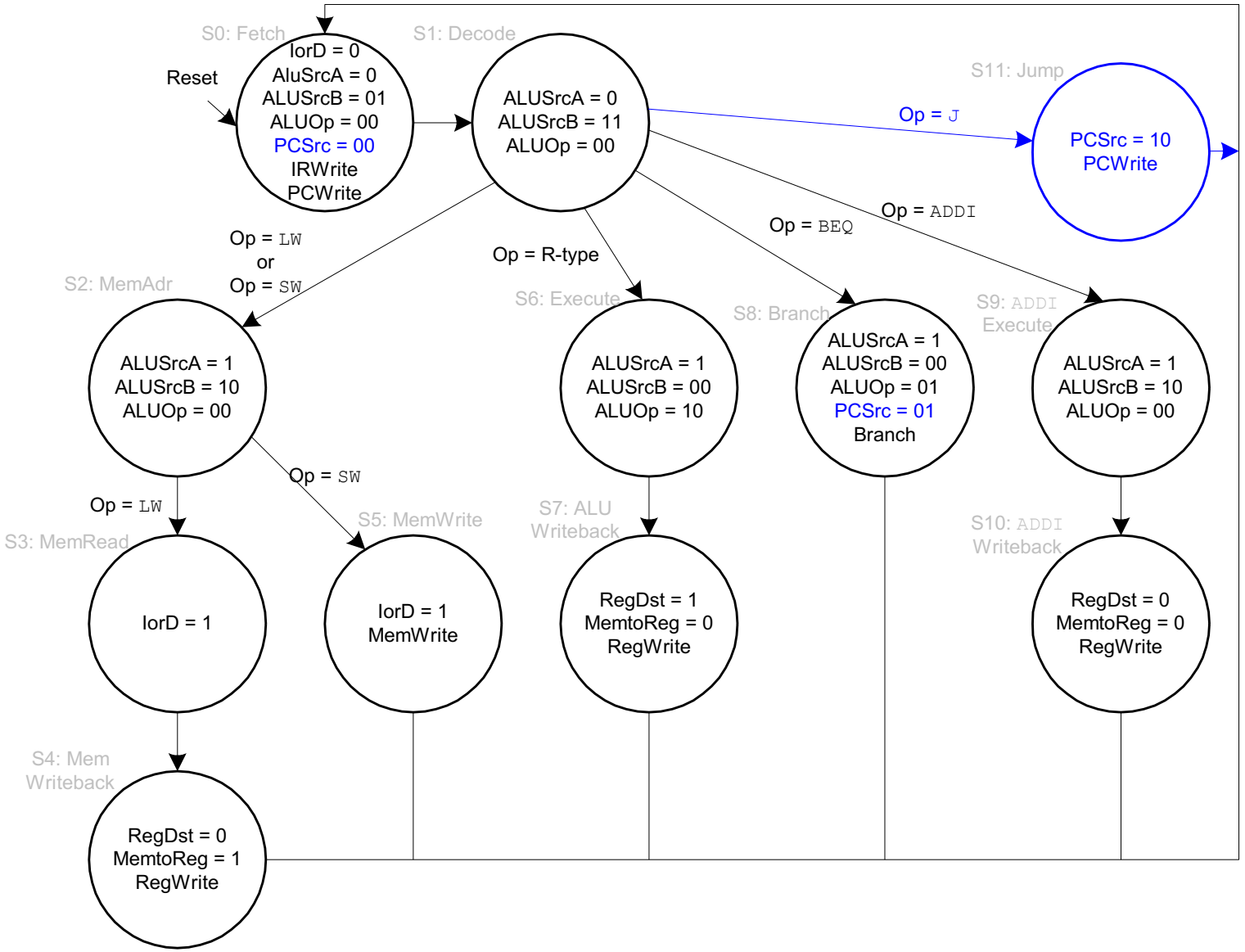
# Extended Functionality: j



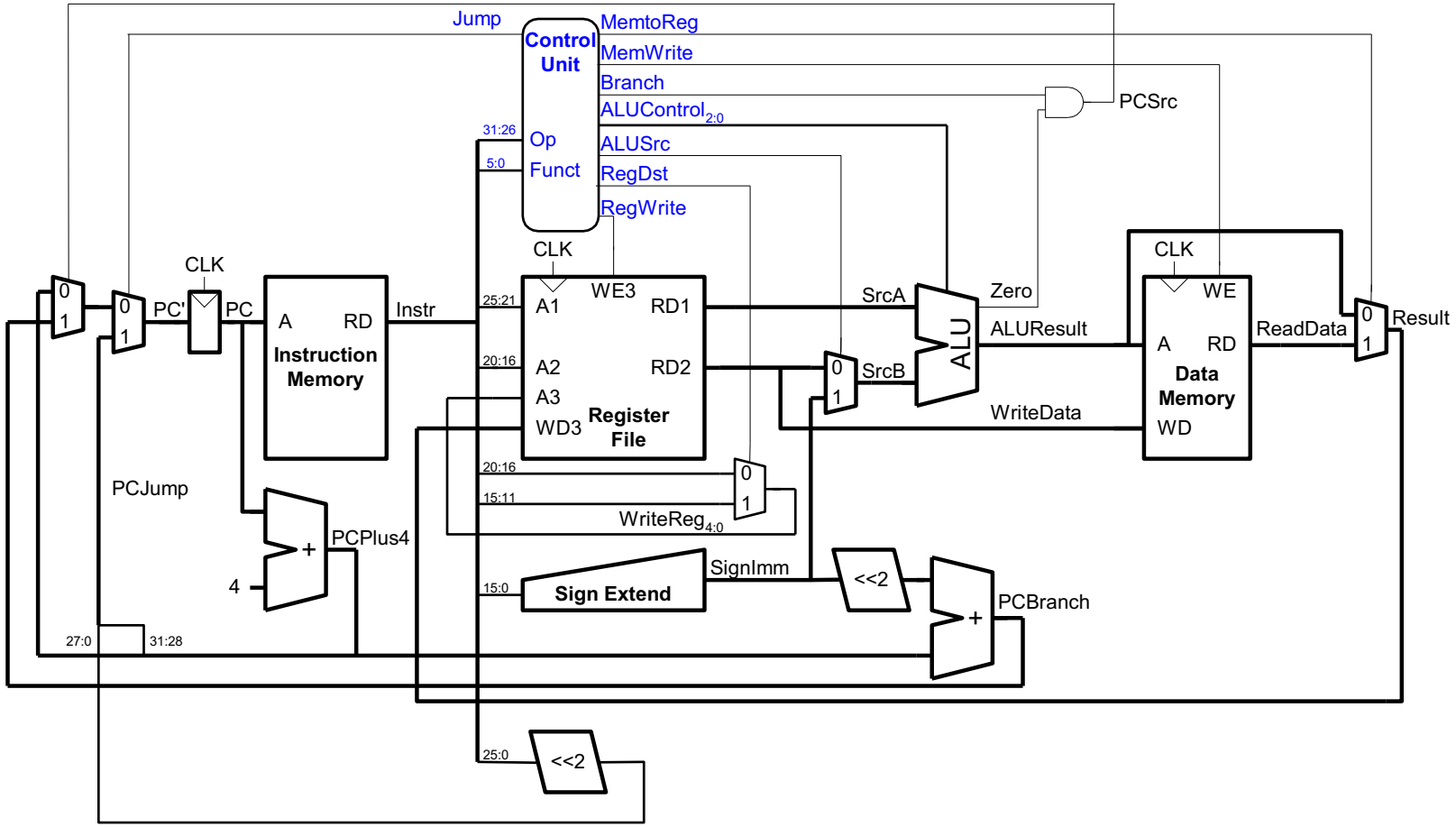
# Control FSM: `j`



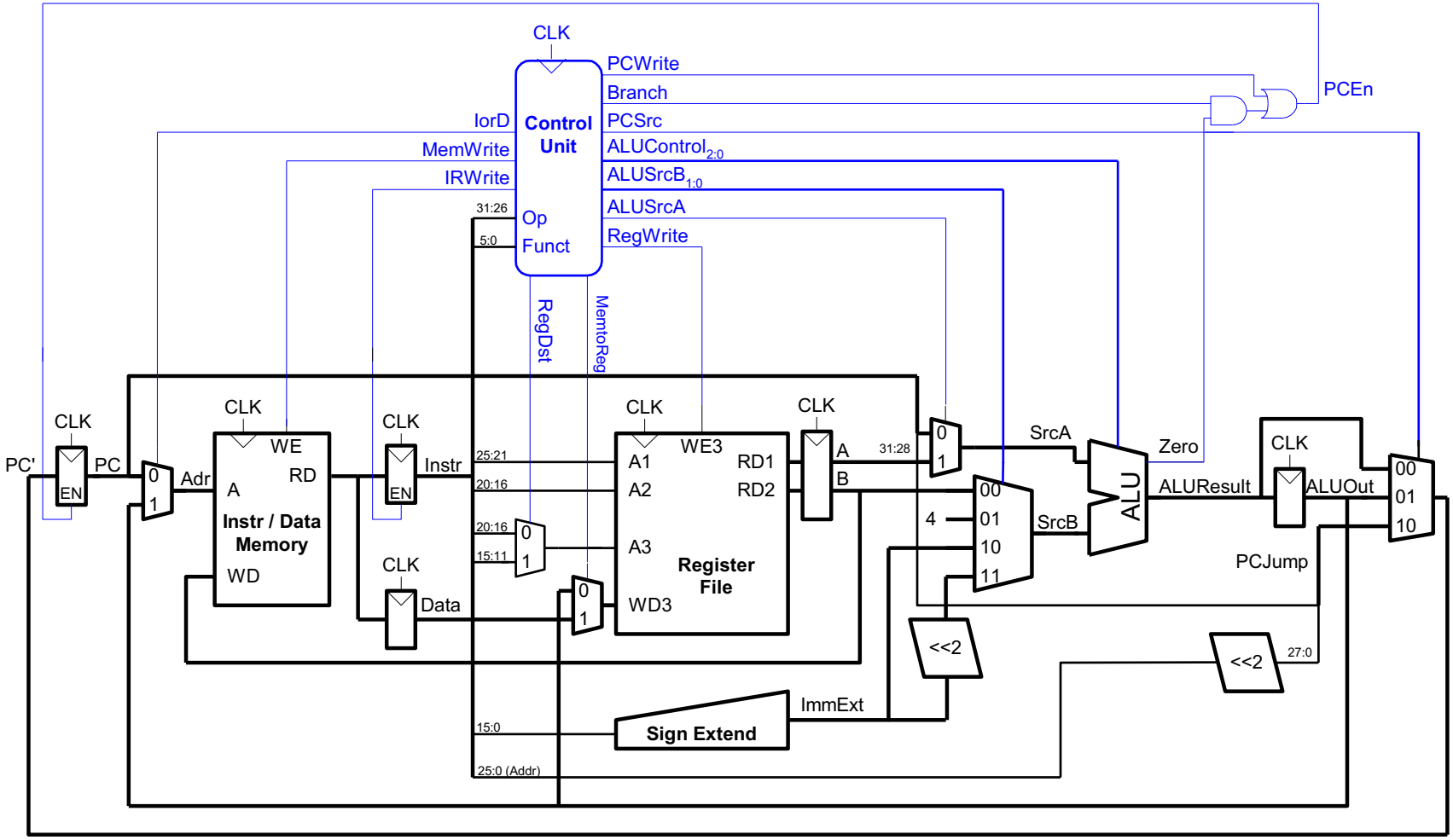
# Control FSM: j



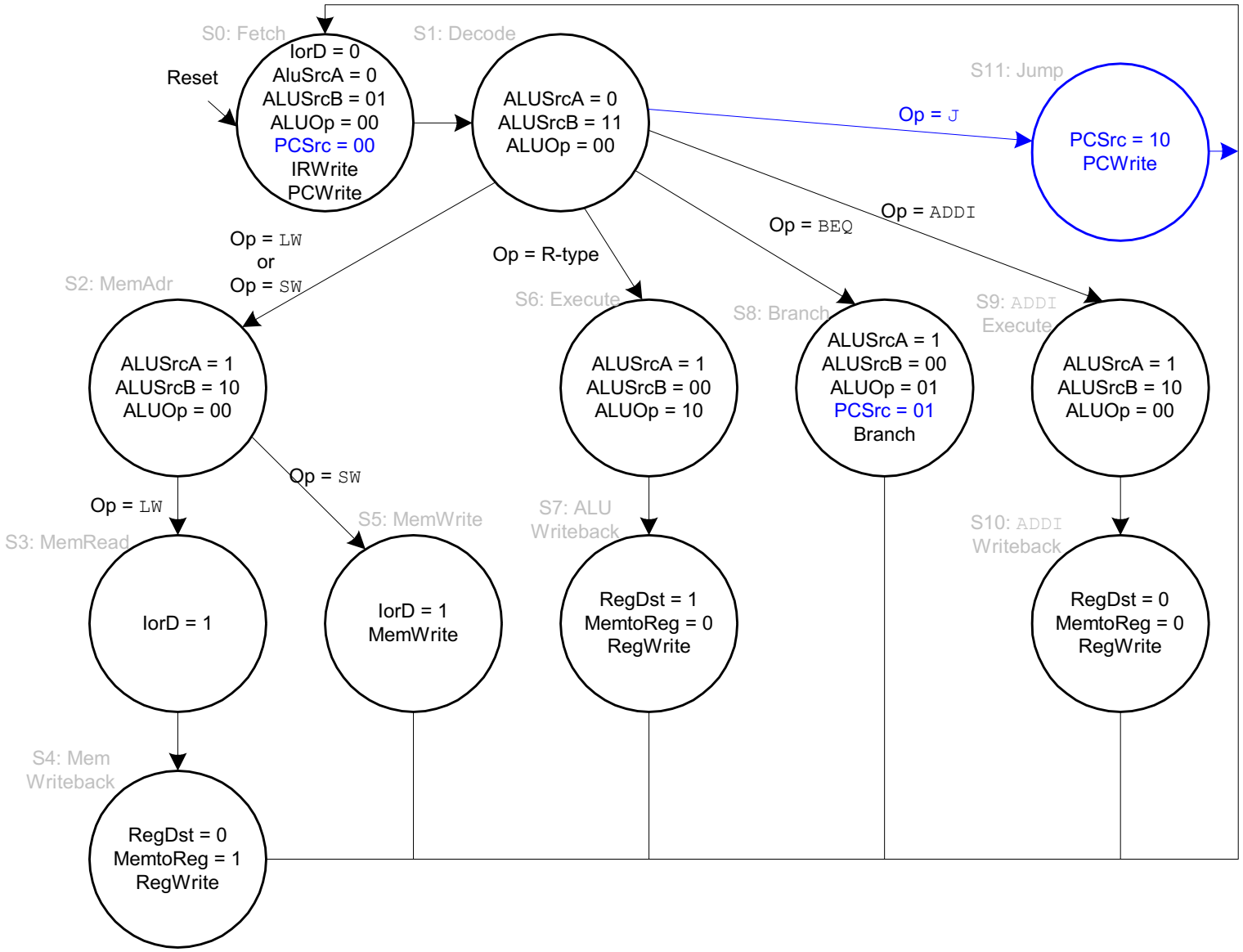
# Review: Single-Cycle MIPS Processor



# Review: Multi-Cycle MIPS Processor



# Review: Multi-Cycle MIPS FSM




**What is the shortcoming of this design?**

**What does this design assume about memory?**

# What If Memory Takes > One Cycle?

- Stay in the same "memory access" state until memory returns the data
- "Memory Ready?" bit is an input to the control logic that determines the next state

# Design of Digital Circuits

## Lecture 12: Microarchitecture II

Prof. Onur Mutlu
ETH Zurich

Spring 2018

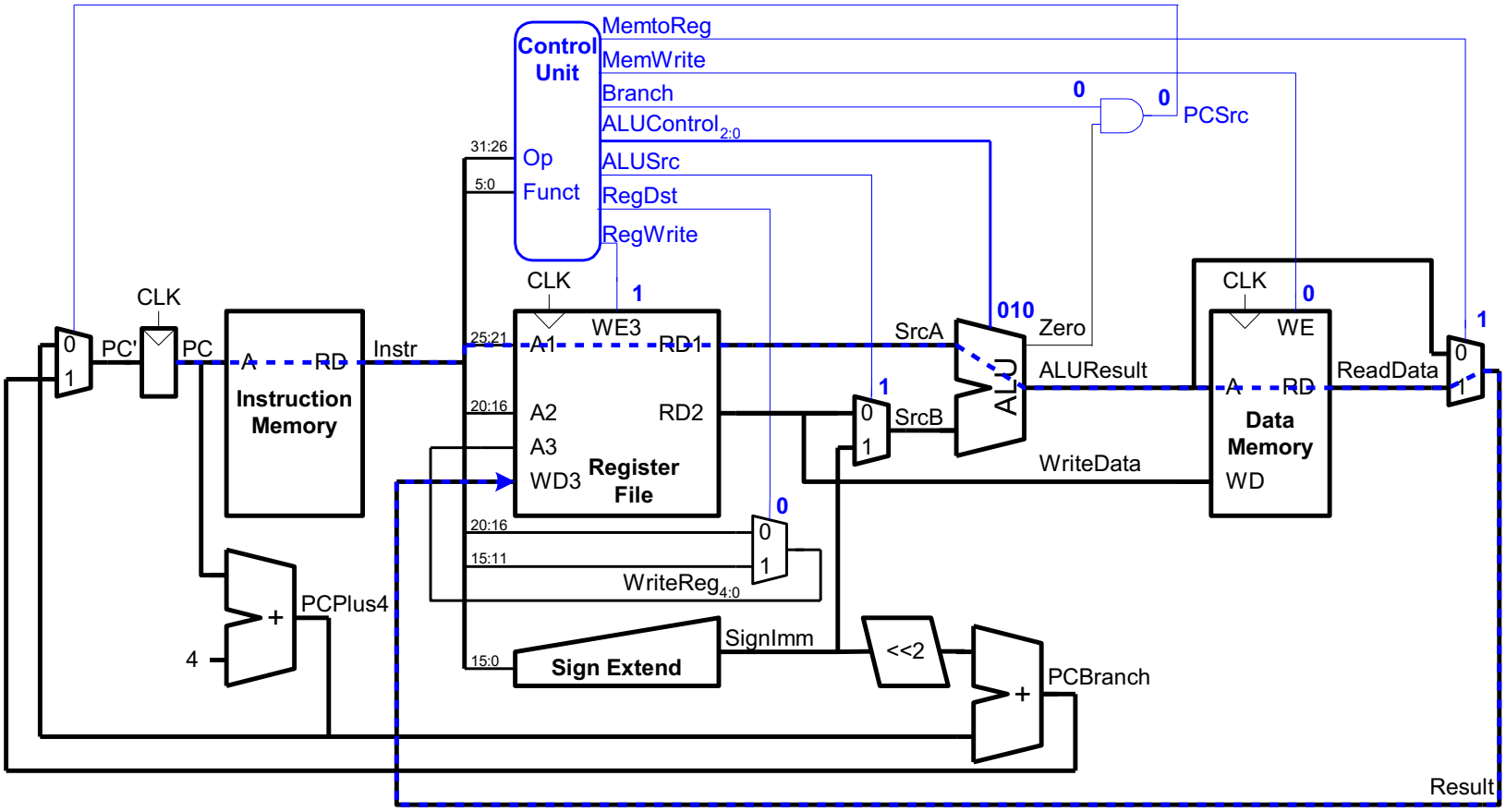
29 March 2019

We did not cover the following slides in lecture.
These are to reinforce your understanding.
The slides are mainly based on your textbook.

# More on Performance Analysis

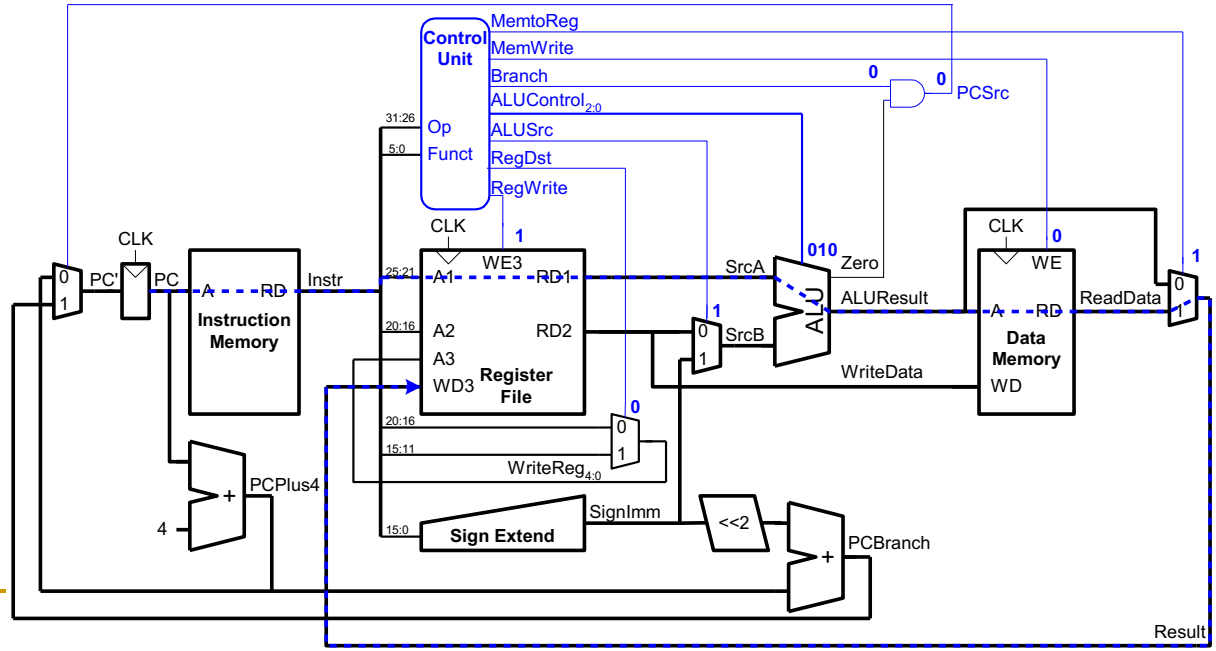
# Single-Cycle Performance

- $T_C$ is limited by the critical path (`lw`)



# Single-Cycle Performance

- Single-cycle critical path:
  - $T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$
- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$



# Single-Cycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_PC}$ | 30 |
| Register setup | $t_{setup}$ | 20 |
| Multiplexer | $t_{mux}$ | 25 |
| ALU | $t_{ALU}$ | 200 |
| Memory read | $t_{mem}$ | 250 |
| Register file read | $t_{RFread}$ | 150 |
| Register file setup | $t_{RFsetup}$ | 20 |

$T_c$ =

# Single-Cycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_PC}$ | 30 |
| Register setup | $t_{setup}$ | 20 |
| Multiplexer | $t_{mux}$ | 25 |
| ALU | $t_{ALU}$ | 200 |
| Memory read | $t_{mem}$ | 250 |
| Register file read | $t_{RFread}$ | 150 |
| Register file setup | $t_{RFsetup}$ | 20 |

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$$
$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$
$$= 925 \text{ ps}$$

# Single-Cycle Performance Example

- Example:

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

# Single-Cycle Performance Example

- Example:

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

***Execution Time*** = # instructions x CPI x $T_c$

= $(100 \times 10^9)(1)(925 \times 10^{-12}\ s)$

= 92.5 seconds

# Multi-Cycle Performance: CPI

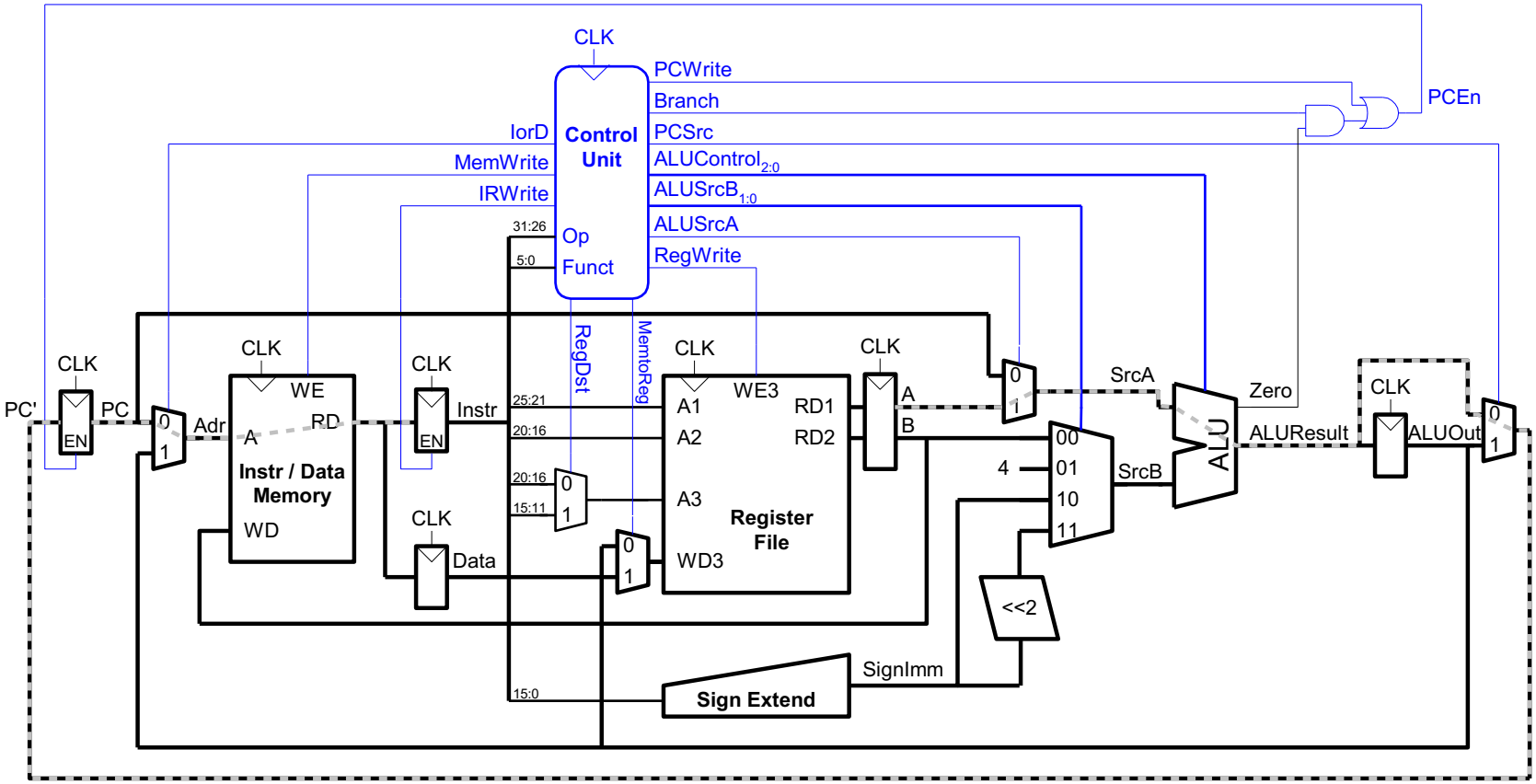
- Instructions take different number of cycles:
  - 3 cycles: `beq, j`
  - 4 cycles: `R-Type, sw, addi`
  - 5 cycles: `lw` **Realistic?**
- CPI is weighted average, e.g. SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

- *Average CPI* = (0.11 + 0.02) 3 +(0.52 + 0.10) 4 +(0.25) 5
  = 4.12

# Multi-cycle Performance: Cycle Time

- Multi-cycle critical path:

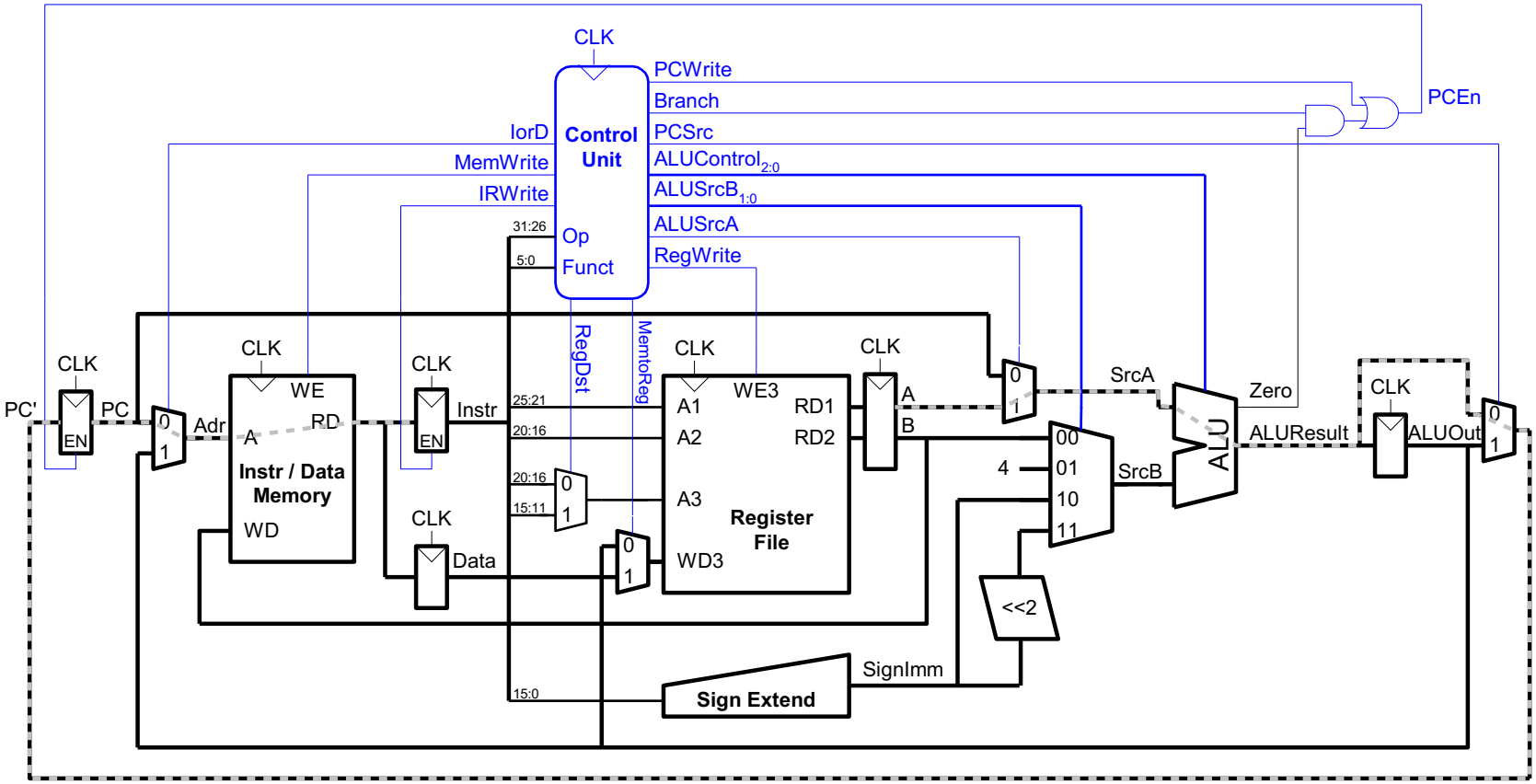
  $T_c$ =



# Multi-cycle Performance: Cycle Time

- Multi-cycle critical path:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



# Multi-Cycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_PC}$ | 30 |
| Register setup | $t_{setup}$ | 20 |
| Multiplexer | $t_{mux}$ | 25 |
| ALU | $t_{ALU}$ | 200 |
| Memory read | $t_{mem}$ | 250 |
| Register file read | $t_{RFread}$ | 150 |
| Register file setup | $t_{RFsetup}$ | 20 |

$T_c$ =

# Multi-Cycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_PC}$ | 30 |
| Register setup | $t_{setup}$ | 20 |
| Multiplexer | $t_{mux}$ | 25 |
| ALU | $t_{ALU}$ | 200 |
| Memory read | $t_{mem}$ | 250 |
| Register file read | $t_{RFread}$ | 150 |
| Register file setup | $t_{RFsetup}$ | 20 |

$T_c = t_{pcq\_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$

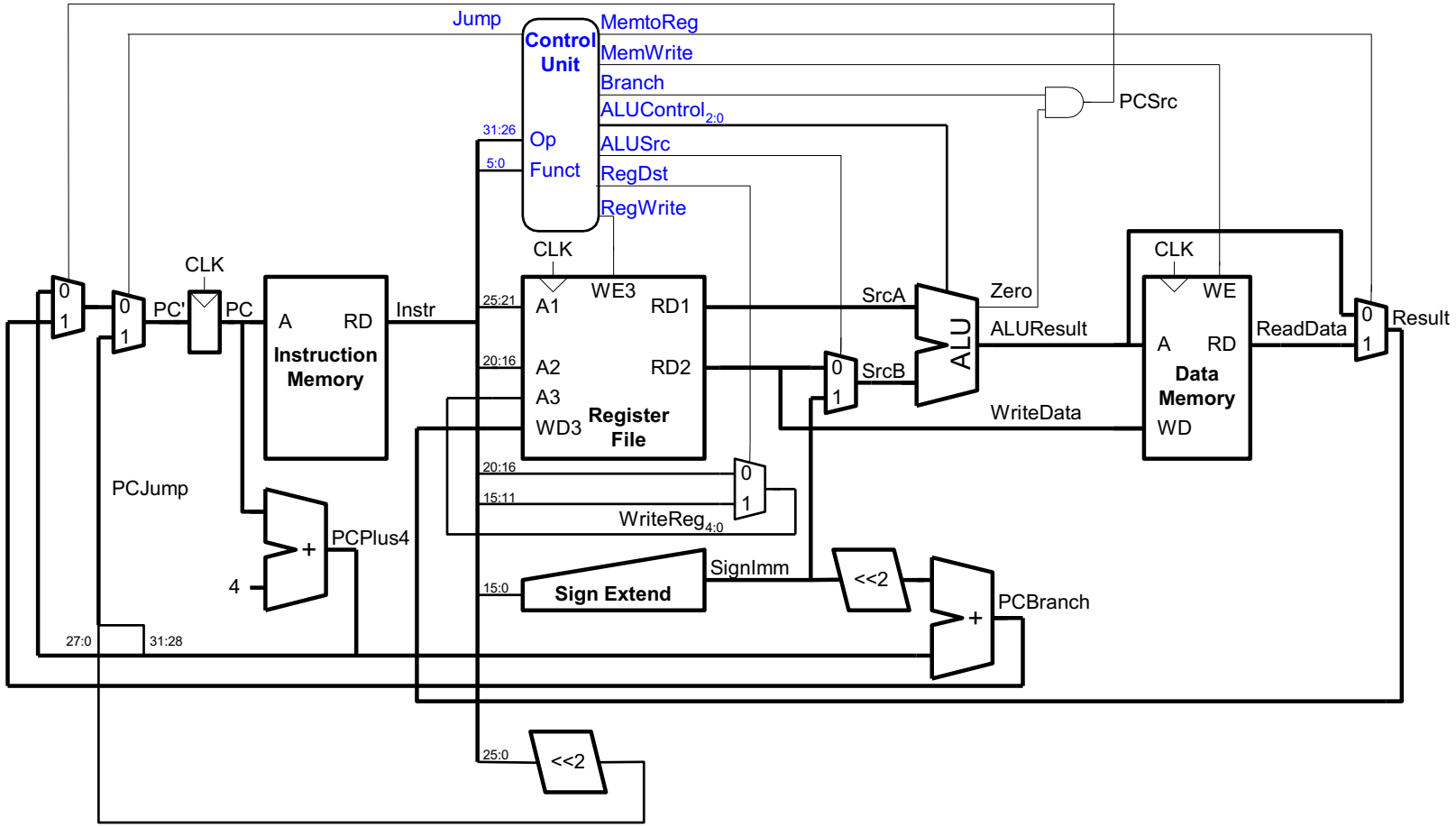
$= [30 + 25 + 250 + 20]$ ps

$= 325$ ps

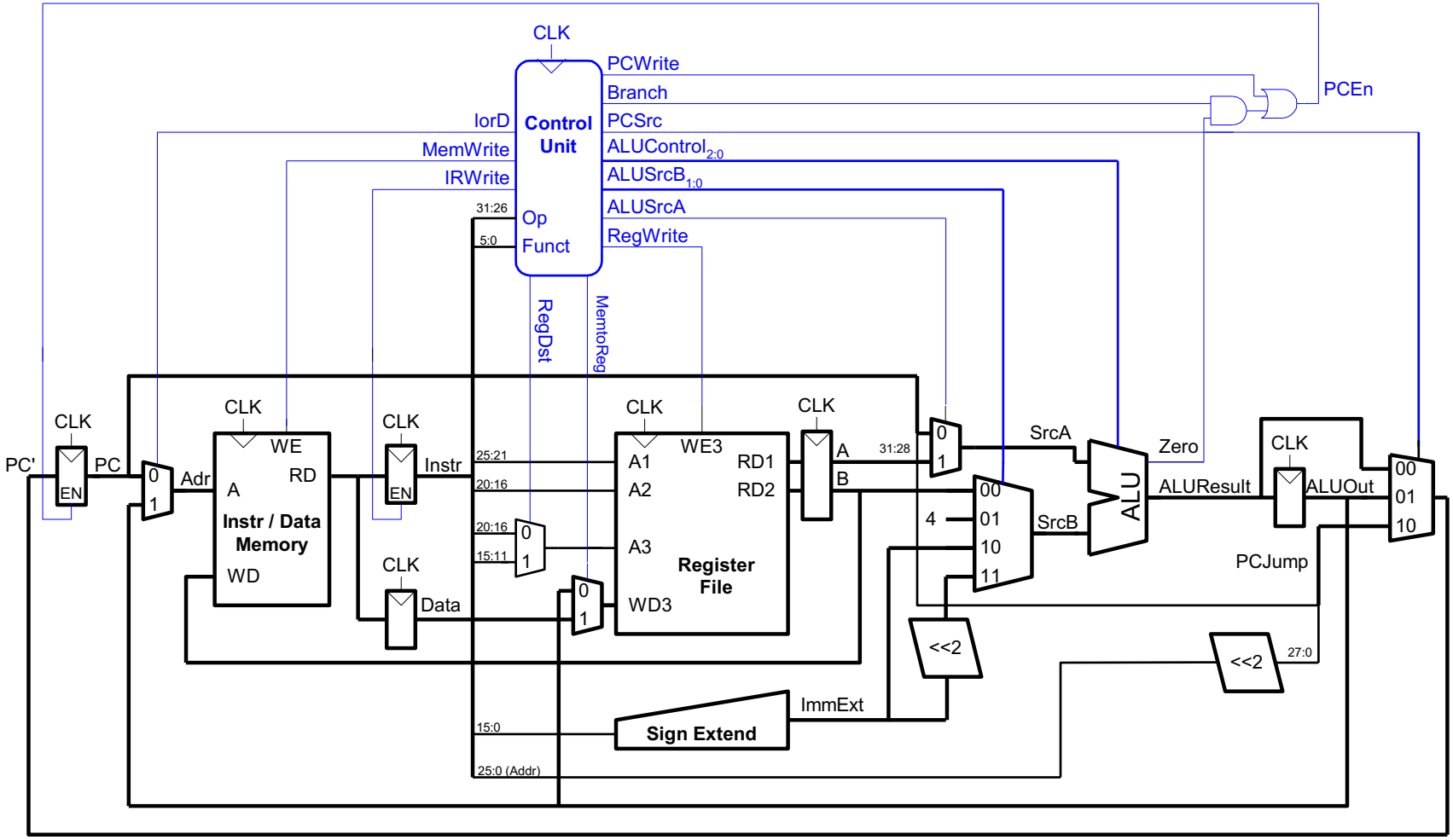
# Multi-Cycle Performance Example

- For a program with 100 billion instructions executing on a multi-cycle MIPS processor
  - CPI = 4.12
  - $T_c$ = 325 ps
- *Execution Time* = (# instructions) × CPI × $T_c$
  = $(100 \times 10^9)(4.12)(325 \times 10^{-12})$
  = 133.9 seconds
- This is slower than the single-cycle processor (92.5 seconds). Why?
- Did we break the stages in a balanced manner?
- Overhead of register setup/hold paid many times
- How would the results change with different assumptions on memory latency and instruction mix?

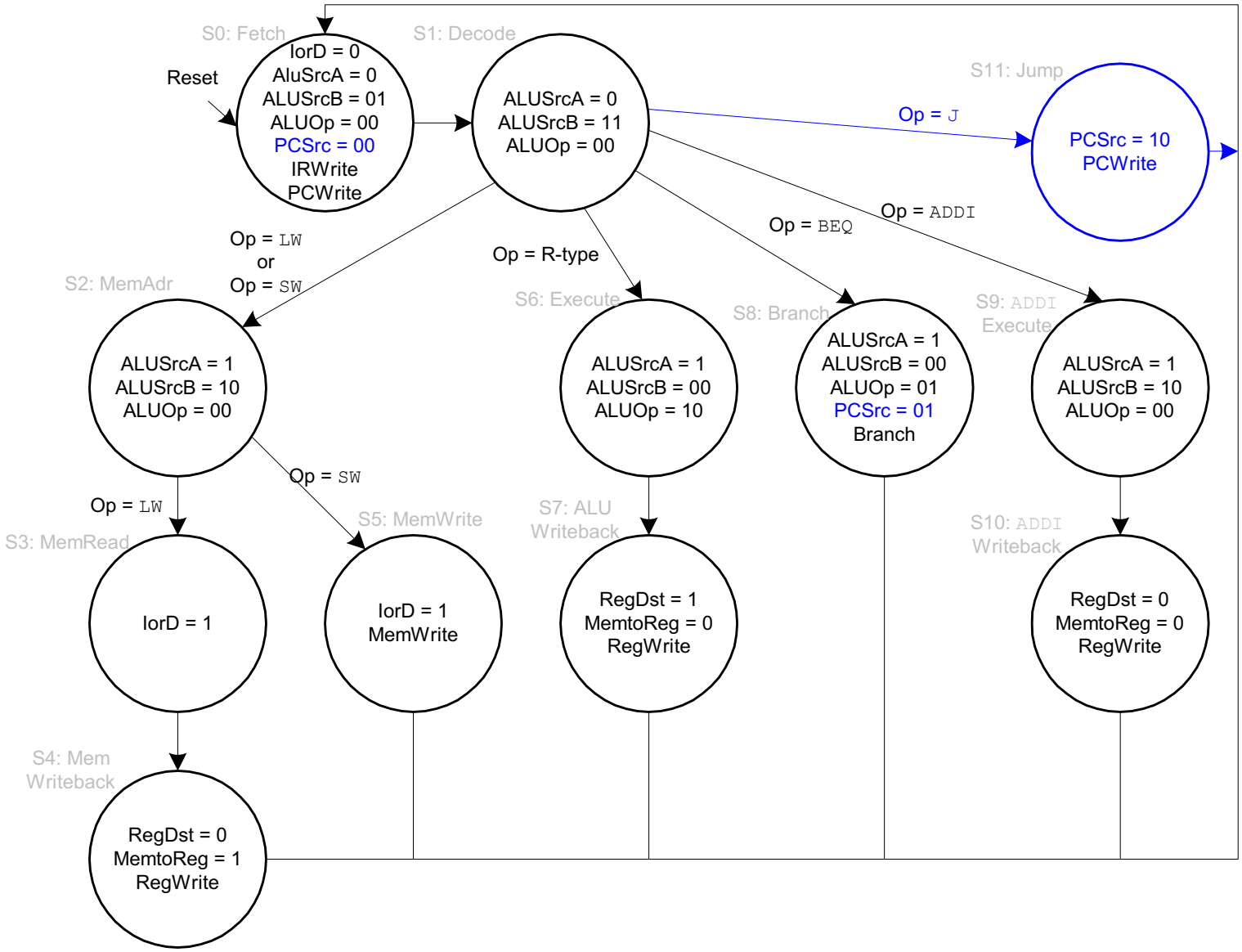
# Review: Single-Cycle MIPS Processor



# Review: Multi-Cycle MIPS Processor



# Review: Multi-Cycle MIPS FSM




**What is the shortcoming of this design?**

**What does this design assume about memory?**

# What If Memory Takes > One Cycle?

- Stay in the same "memory access" state until memory returns the data
- "Memory Ready?" bit is an input to the control logic that determines the next state

# Backup Slides on Single-Cycle Uarch for Your Own Study

Please study these to reinforce the concepts
we covered in lectures.

Please do the readings together with these slides:
H&H, Chapter 7.1-7.3, 7.6

# Another Single-Cycle MIPS Processor (from H&H)

These are slides for your own study.
They are to complement your reading
H&H, Chapter 7.1-7.3, 7.6

# What to do with the Program Counter?

- The PC needs to be incremented by 4 during each cycle (for the time being).
- Initial PC value (after reset) is `0x00400000`

```
reg [31:0] PC_p, PC_n;      // Present and next state of PC

// […]

  assign PC_n <= PC_p + 4;                    // Increment by 4;

  always @ (posedge clk, negedge rst)
    begin
       if (rst == '0') PC_p <= 32'h00400000; // default
       else            PC_p <= PC_n;         // when clk
    end
```

# We Need a Register File

- **Store 32 registers, each 32-bit**
  - $2^5$ == 32, we need 5 bits to address each
- **Every R-type instruction uses 3 register**
  - Two for reading (RS, RT)
  - One for writing (RD)
- **We need a special memory with:**
  - 2 read ports (address x2, data out x2)
  - 1 write port (address, data in)

# Register File

```
input [4:0]   a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input         we_rd;
output [31:0] do_rs, do_rt;

  reg [31:0] R_arr [31:0]; // Array that stores regs

  // Circuit description
  assign do_rs = R_arr[a_rs];          // Read RS


  assign do_rt = R_arr[a_rt];          // Read RT


  always @ (posedge clk)
      if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

# Register File

```
input [4:0]   a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input         we_rd;
output [31:0] do_rs, do_rt;

  reg [31:0] R_arr [31:0]; // Array that stores regs

  // Circuit description; add the trick with $0
  assign do_rs = (a_rs != 5'b00000)?   // is address 0?
                  R_arr[a_rs] : 0;     // Read RS or 0

  assign do_rt = (a_rt != 5'b00000)?   // is address 0?
                  R_arr[a_rt] : 0;     // Read RT or 0

  always @ (posedge clk)
      if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

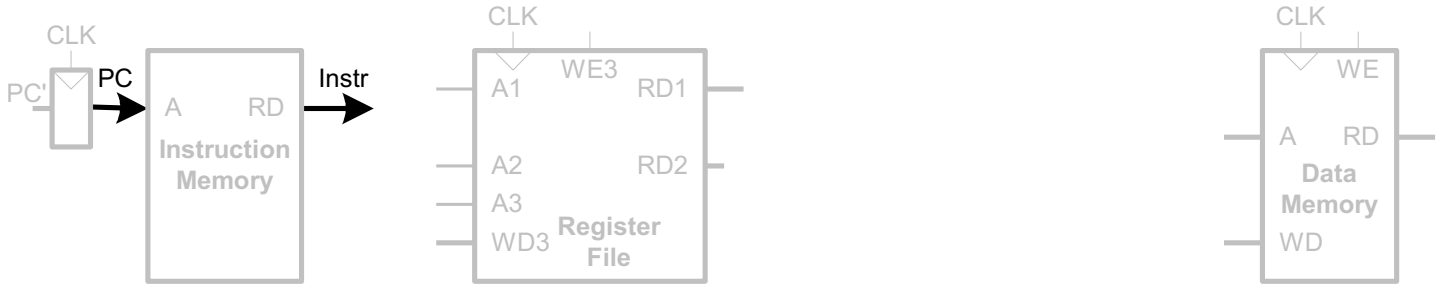
# Data Memory Example

- **Will be used to store the bulk of data**

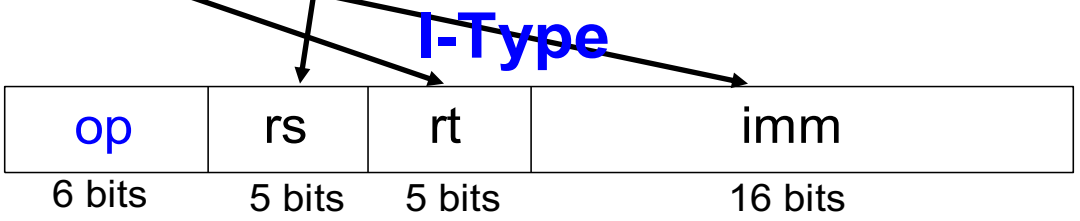
```
input [15:0]  addr; // Only 16 bits in this example
input [31:0]  di;
input         we;
output [31:0] do;

  reg [31:0] M_arr [0:65535];          // Array for Memory

  // Circuit description
  assign do = M_arr[addr];             // Read memory

  always @ (posedge clk)
      if (we) M_arr[addr] <= di;       // write memory
```

# Single-Cycle Datapath: lw fetch

- ***STEP 1:*** **Fetch instruction**

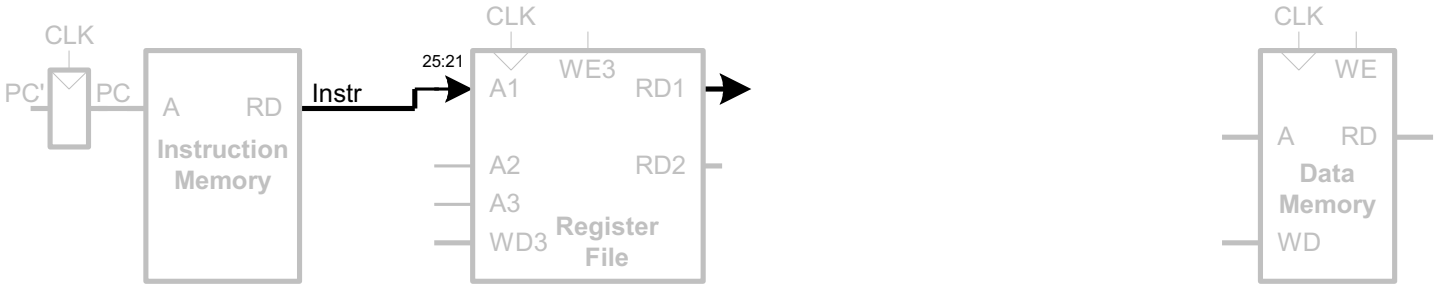



```
lw $s3, 1($0)   # read memory word 1 into $s3
```



# Single-Cycle Datapath: lw register read

- ***STEP 2:*** **Read source operands from register file**



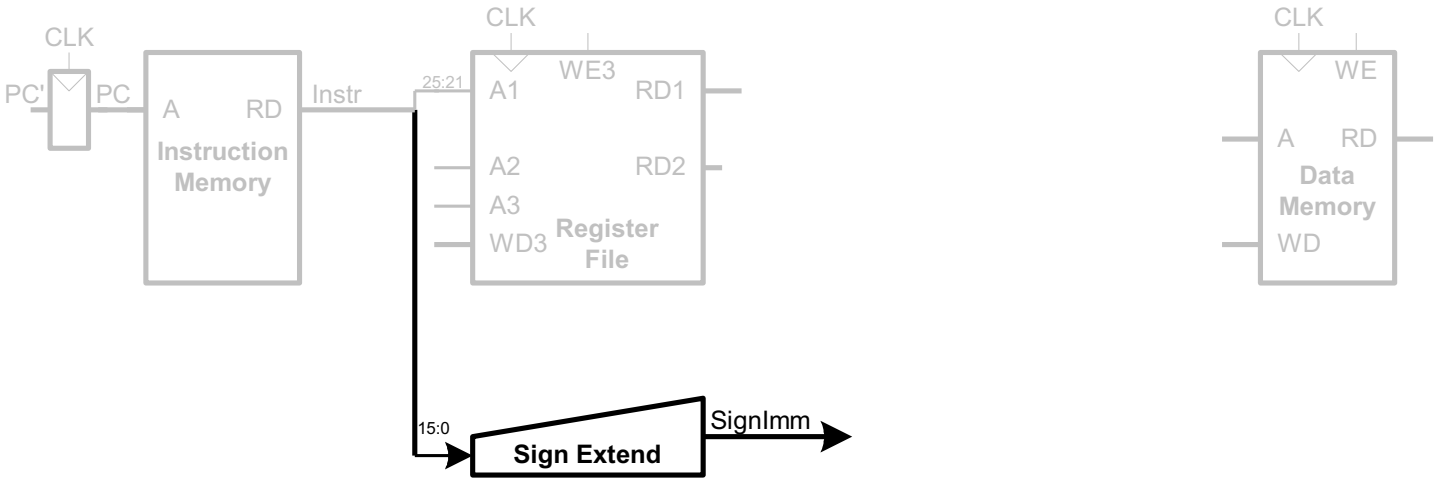

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: lw immediate

- ***STEP 3:*** **Sign-extend the immediate**



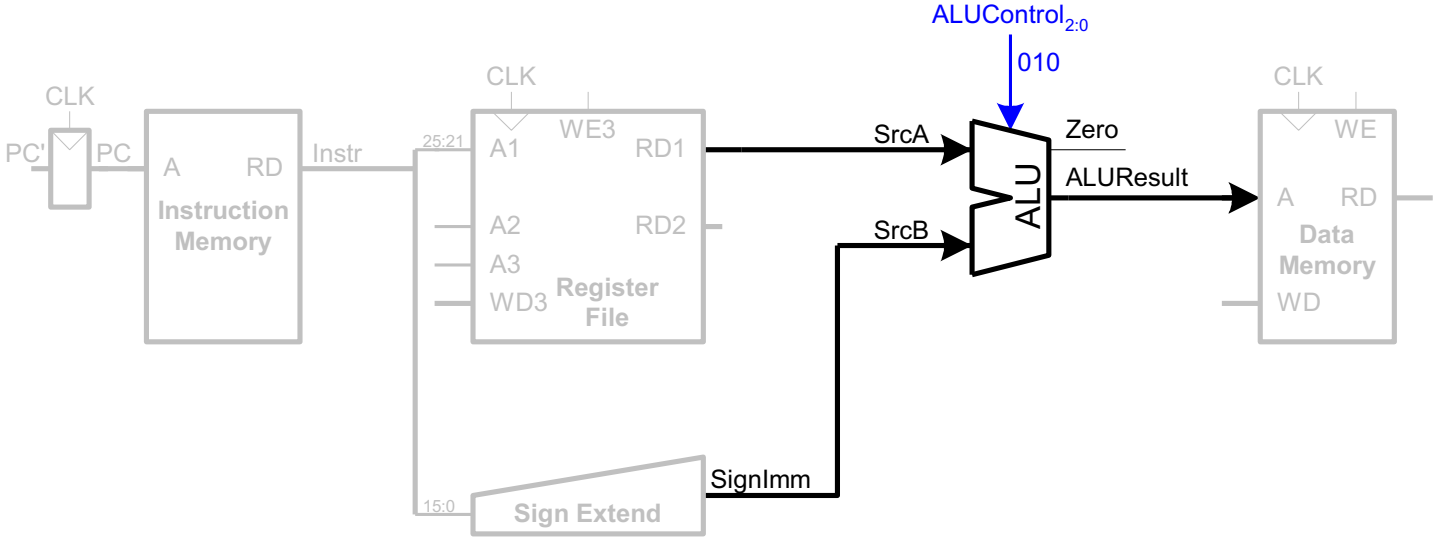

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: lw address

- ***STEP 4:*** **Compute the memory address**



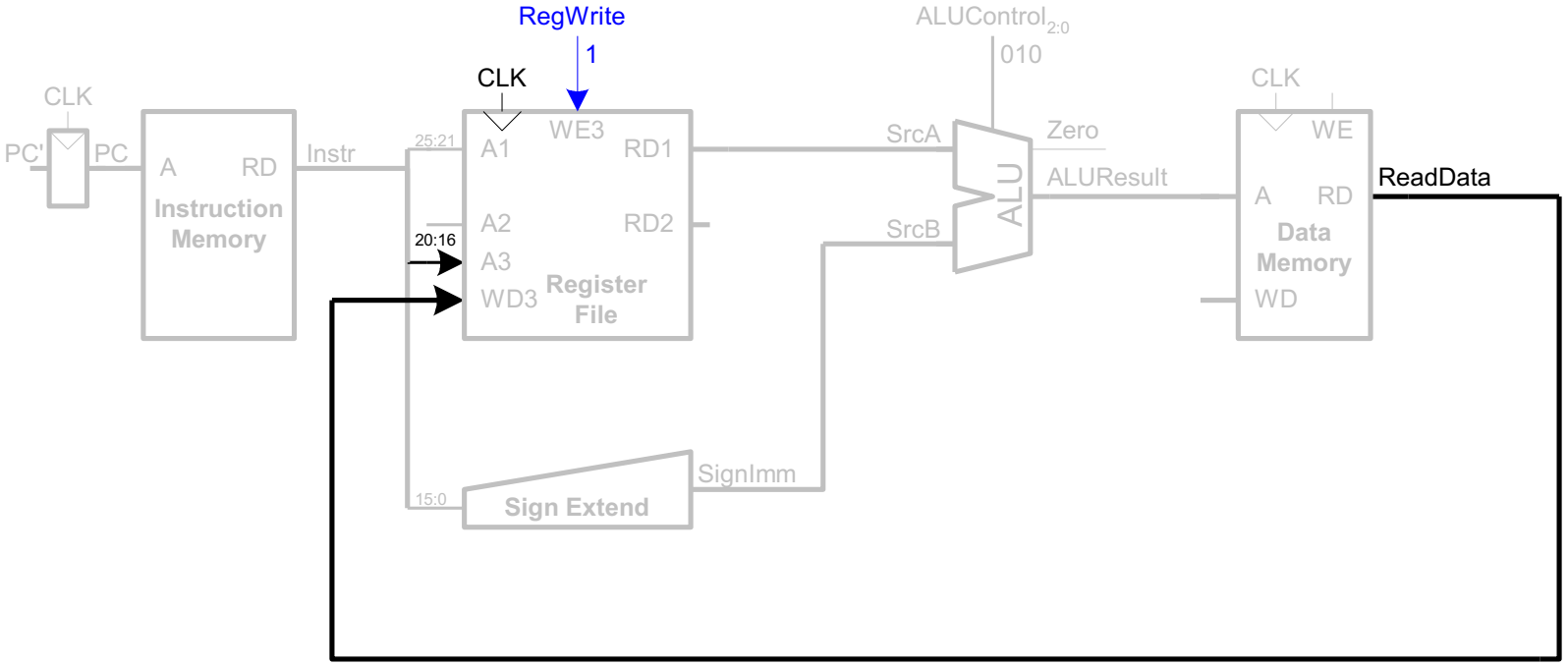

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: lw memory read

- ***STEP 5:*** **Read from memory and write back to register file**



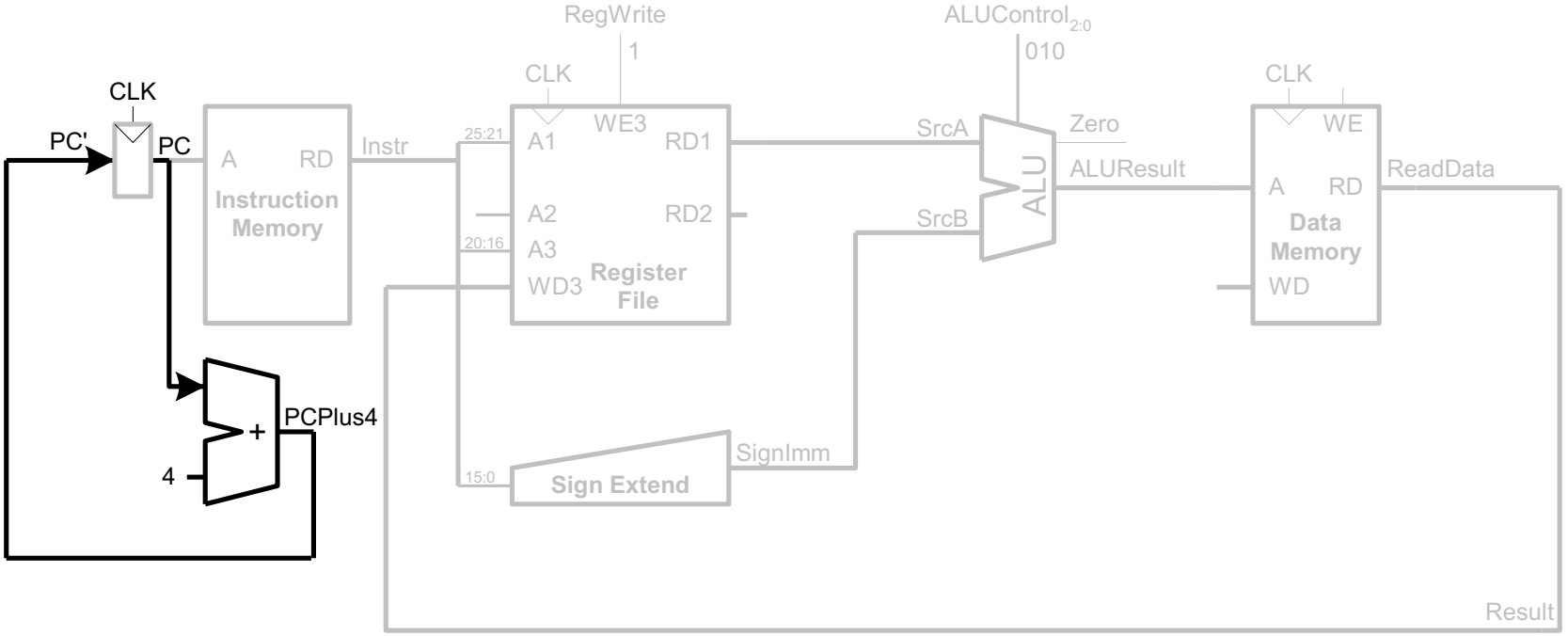

```
lw $s3, 1($0)   # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: lw PC increment

- ***STEP 6:*** **Determine address of next instruction**



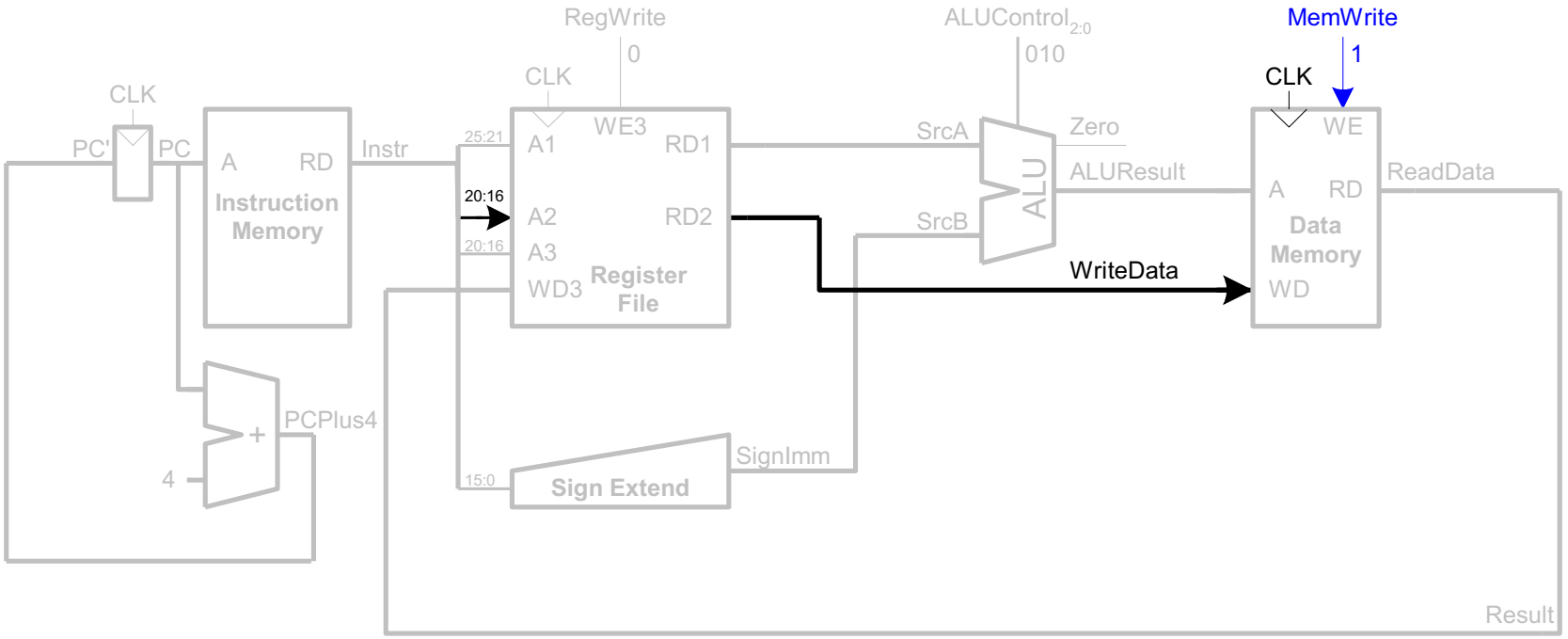

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: sw

- **Write data in rt to memory**



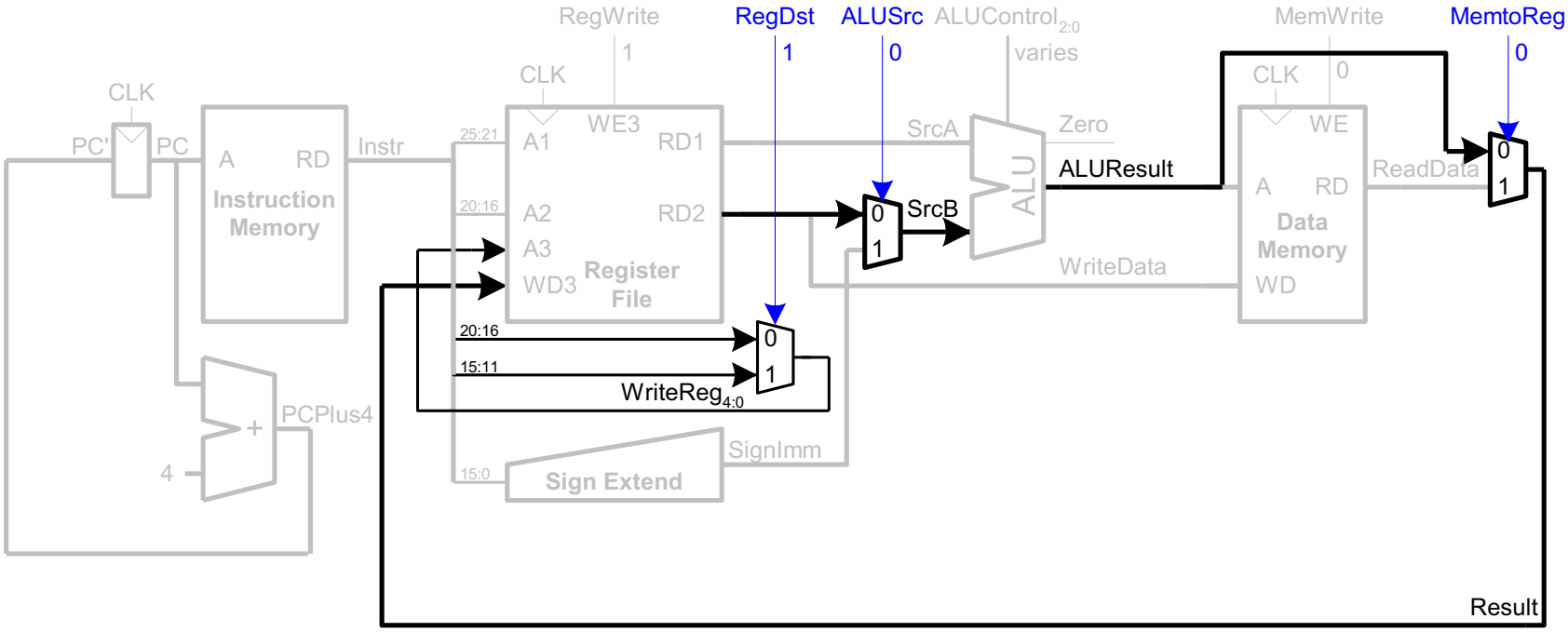

```
sw $t7, 44($0)  # write t7 into memory address 44
```

**I-Type**

| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Single-Cycle Datapath: R-type Instructions

- **Read from rs and rt, write ALUResult to register file**

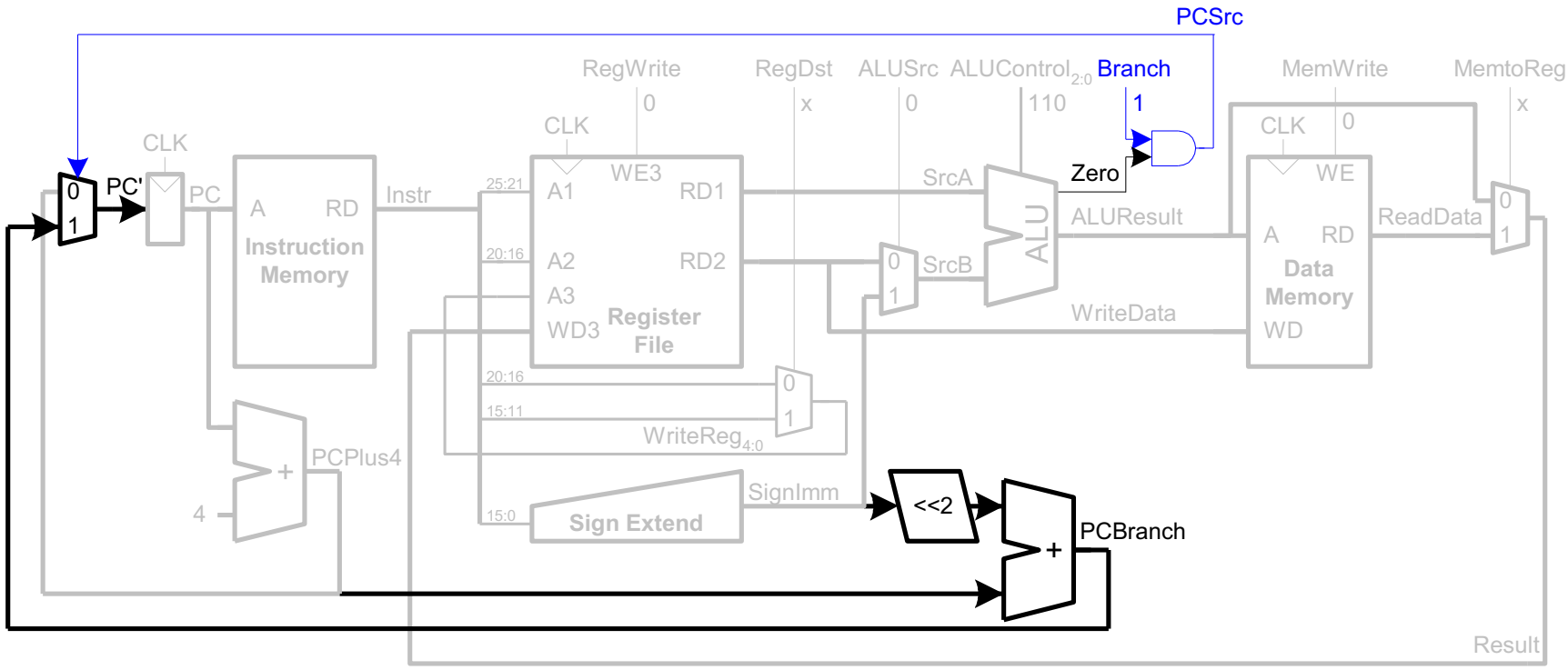



```
add t, b, c  # t = b + c
```

**R-Type**

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

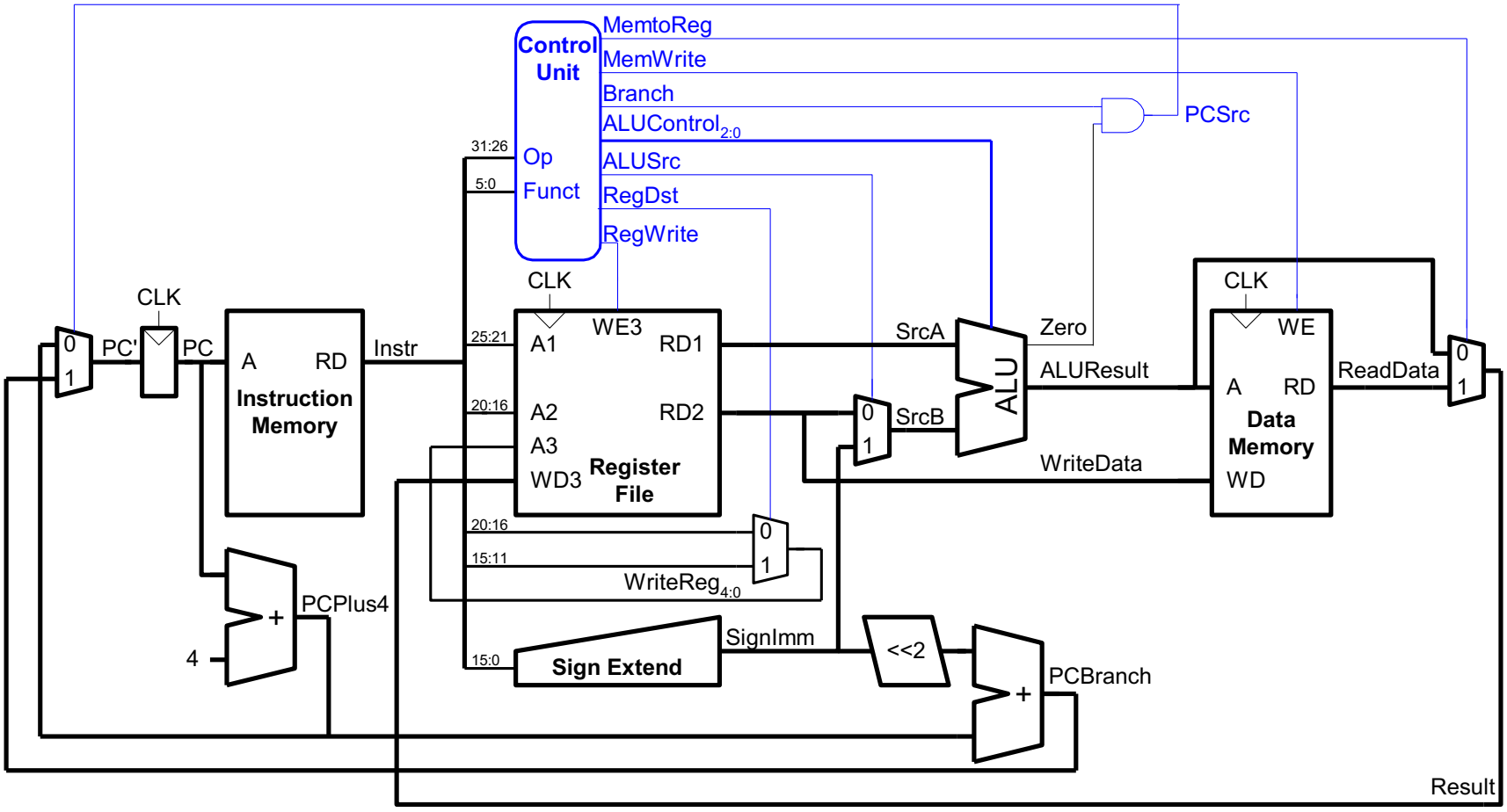
# Single-Cycle Datapath: beq




```
beq  $s0, $s1, target  # branch is taken
```

- **Determine whether values in rs and rt are equal**
  **Calculate BTA = (sign-extended immediate << 2) + (PC+4)**

# Complete Single-Cycle Processor

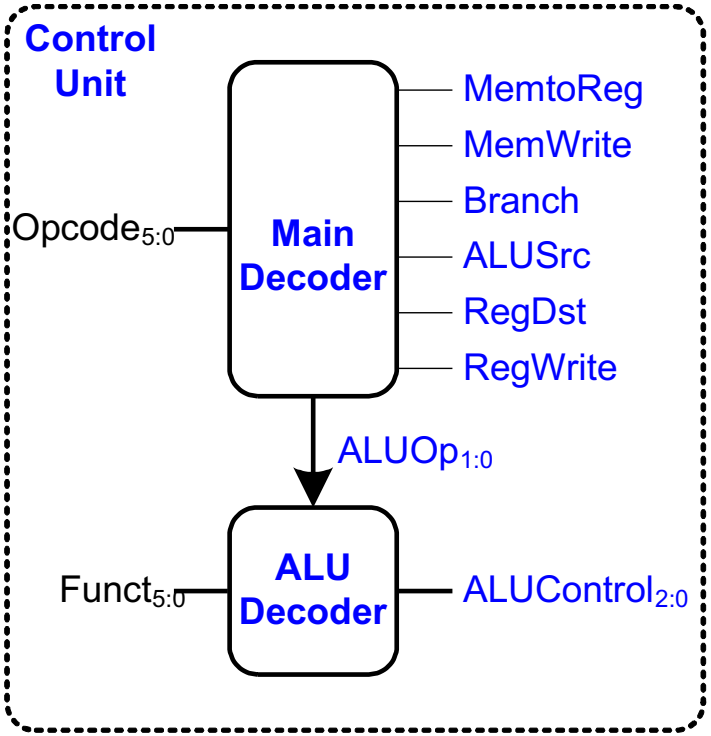


# Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate ***(MUX)***
- **Write Address of Register File**
  - Either RD or RT ***(MUX)***
- **Write Data In of Register File**
  - Either ALU out or Data Memory Out ***(MUX)***
- **Write enable of Register File**
  - Not always a register write ***(MUX)***
- **Write enable of Memory**
  - Only when writing to memory (sw) ***(MUX)***

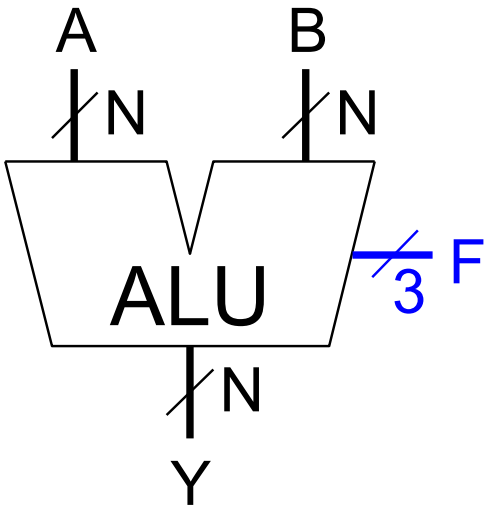
***All these options are our control signals***

# Control Unit



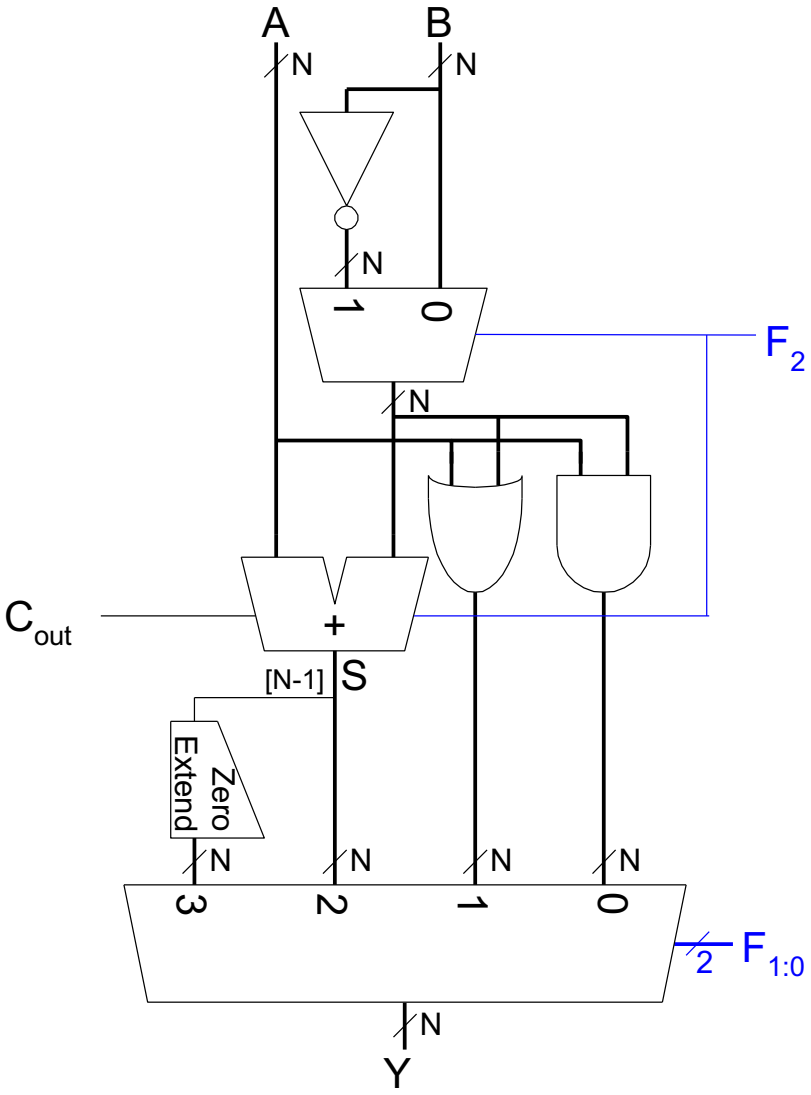

| ALUOp | Meaning |
|---|---|
| 00 | add |
| 01 | subtract |
| 10 | look at `funct` field |
| 11 | n/a |

# ALU Does the Real Work in a Processor



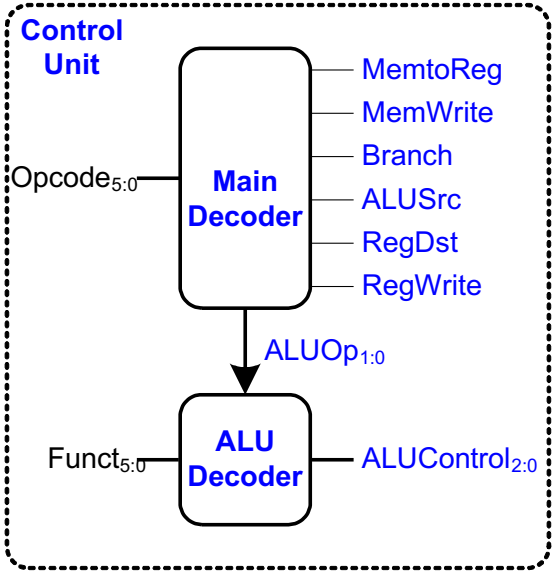

| $F_{2:0}$ | Function |
|---|---|
| 000 | A & B |
| 001 | A \| B |
| 010 | A + B |
| 011 | not used |
| 100 | A & ~B |
| 101 | A \| ~B |
| 110 | A - B |
| 111 | SLT |

# ALU Internals




| $F_{2:0}$ | Function |
|---|---|
| 000 | A & B |
| 001 | A \| B |
| 010 | A + B |
| 011 | not used |
| 100 | A & ~B |
| 101 | A \| ~B |
| 110 | A - B |
| 111 | SLT |

# Control Unit: ALU Decoder




| $ALUOp_{1:0}$ | Meaning |
|---|---|
| 00 | Add |
| 01 | Subtract |
| 10 | Look at Funct |
| 11 | Not Used |

| $ALUOp_{1:0}$ | Funct | $ALUControl_{2:0}$ |
|---|---|---|
| 00 | X | 010 (Add) |
| X1 | X | 110 (Subtract) |
| 1X | 100000 (add) | 010 (Add) |
| 1X | 100010 (sub) | 110 (Subtract) |
| 1X | 100100 (and) | 000 (And) |
| 1X | 100101 (or) | 001 (Or) |
| 1X | 101010 (slt) | 111 (SLT) |

# Let us Develop our Control Table

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | MemWrite | MemtoReg | ALUOp |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

# Let us Develop our Control Table

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | MemWrite | MemtoReg | ALUOp |
|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | funct |
| | | | | | | | |
| | | | | | | | |

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

# Let us Develop our Control Table

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | MemWrite | MemtoReg | ALUOp |
|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | funct |
| `lw` | 100011 | 1 | 0 | 1 | 0 | 1 | add |
| | | | | | | | |

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

# Let us Develop our Control Table

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | MemWrite | MemtoReg | ALUOp |
|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | funct |
| `lw` | 100011 | 1 | 0 | 1 | 0 | 1 | add |
| `sw` | 101011 | 0 | X | 1 | 1 | X | add |

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

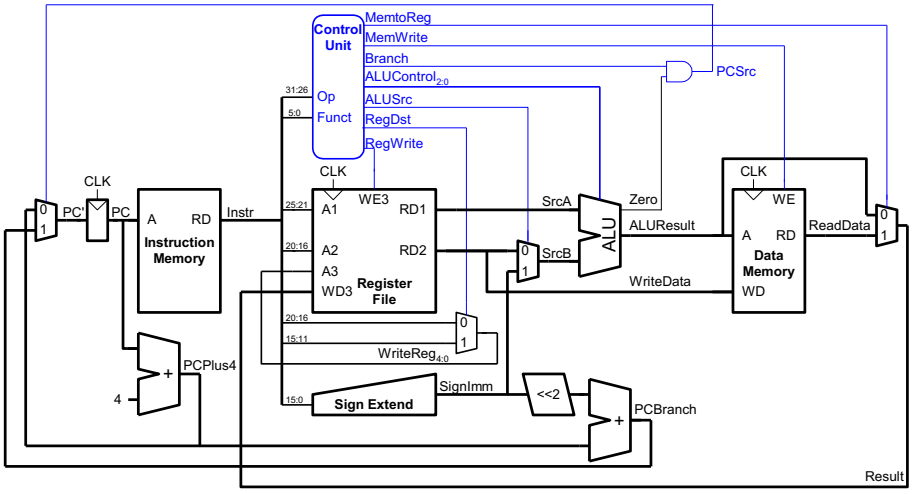
# More Control Signals

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | ALUOp |
|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | funct |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | add |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | add |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | sub |

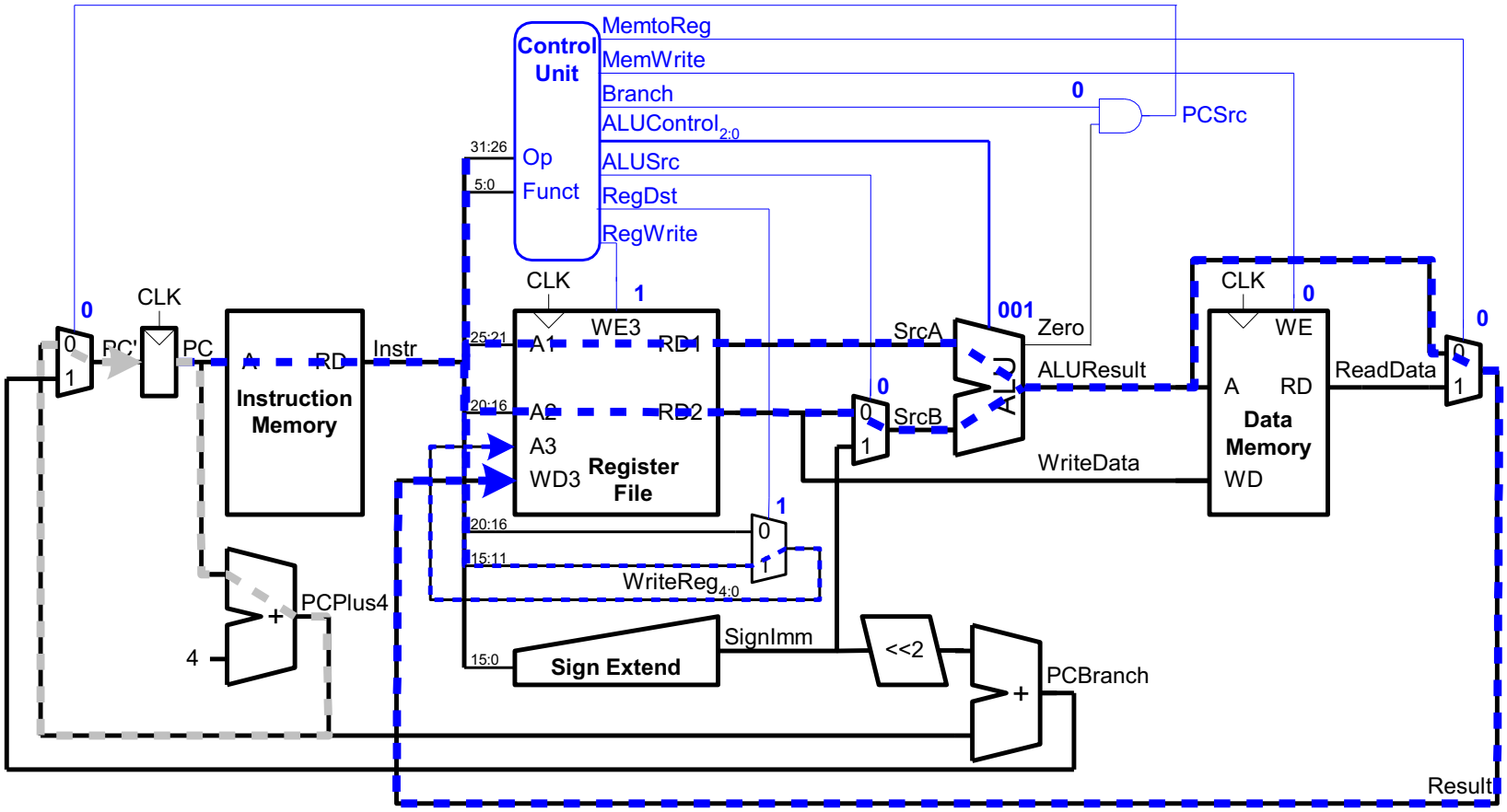
- **New Control Signal**
  - ***Branch***: Are we jumping or not ?

# Control Unit: Main Decoder

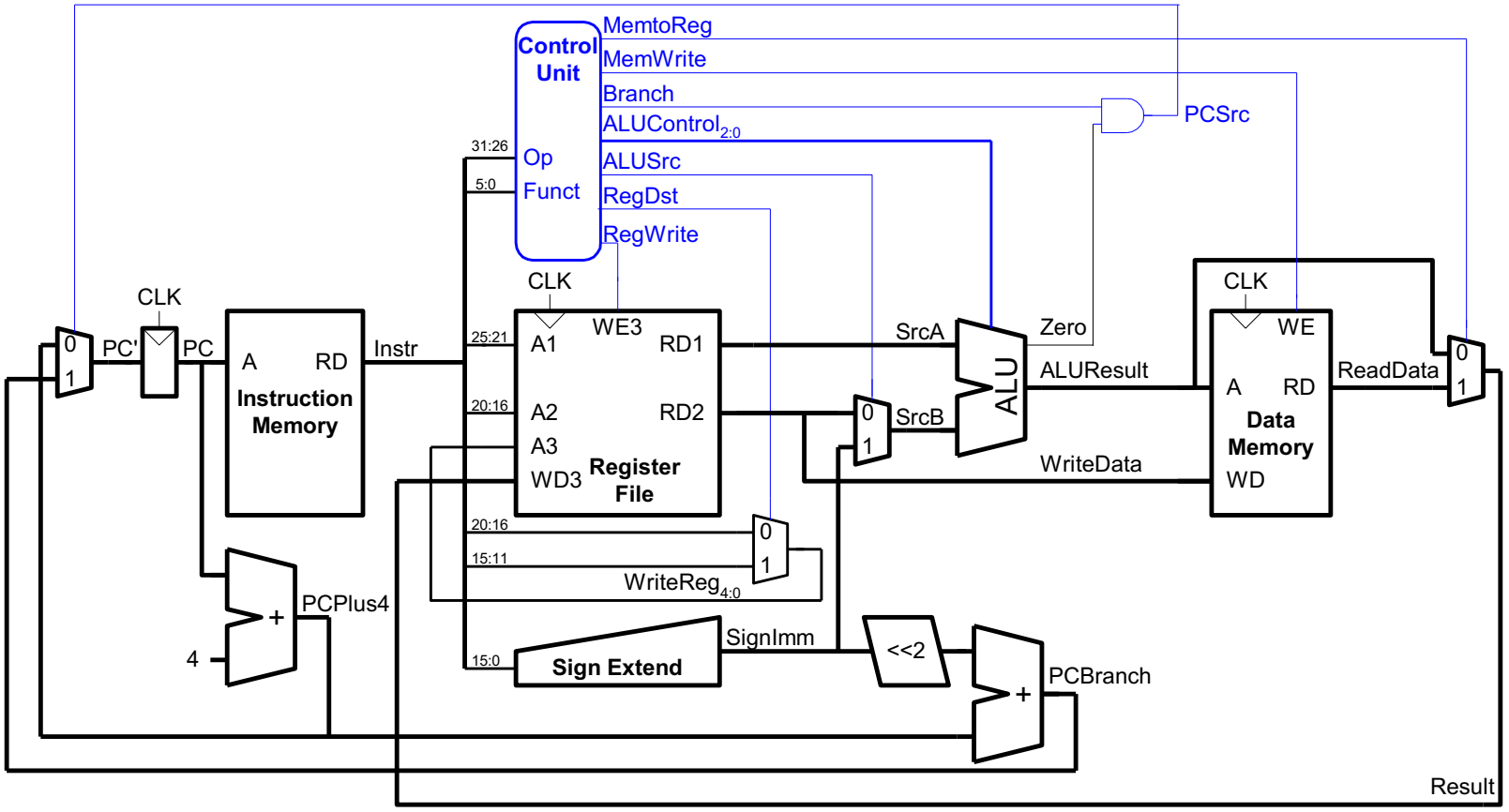
| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ |
|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 |



# Single-Cycle Datapath Example: or



# Extended Functionality: addi

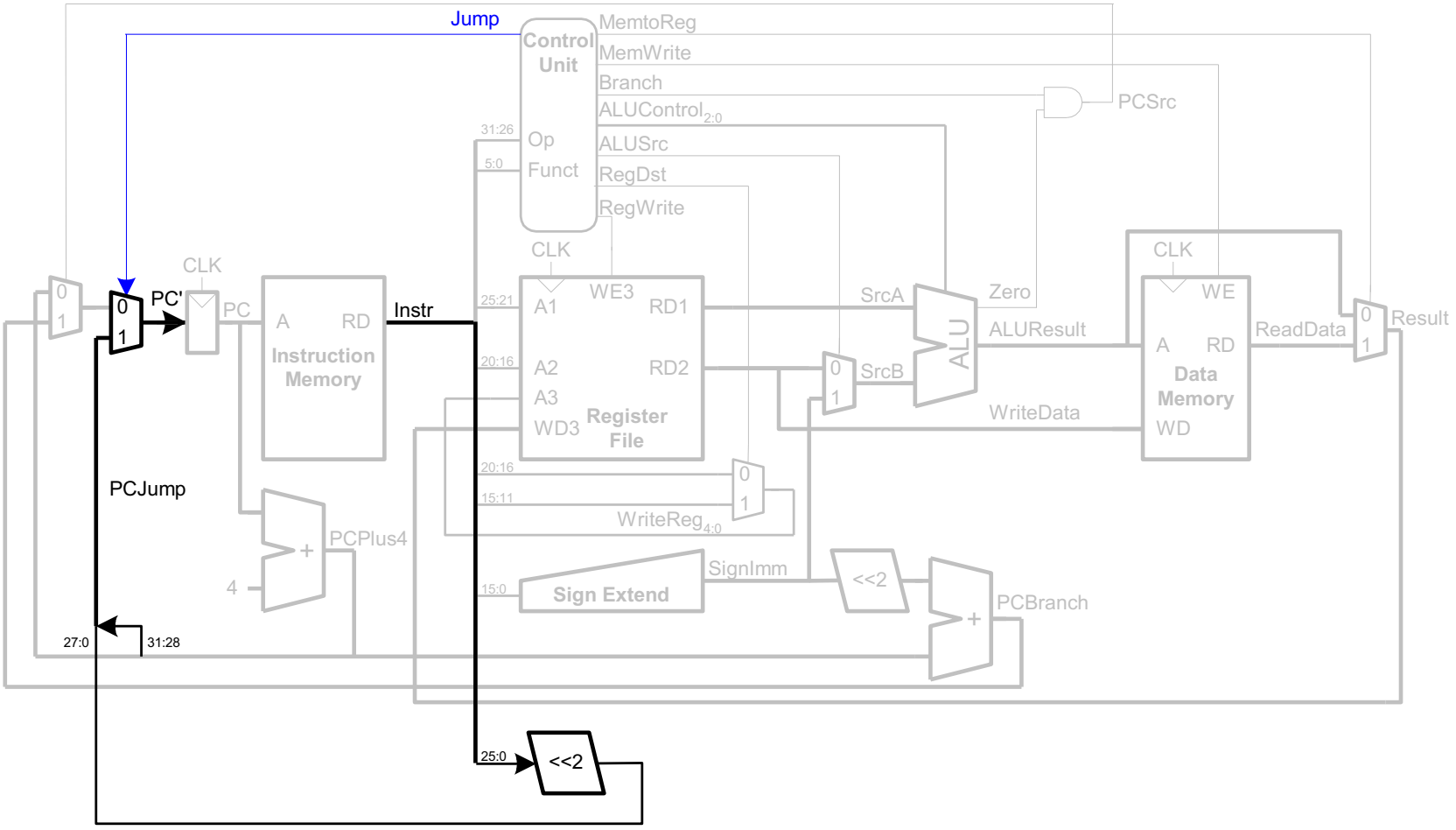



- No change to datapath

# Control Unit: addi

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ |
|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 |
| addi | 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 00 |

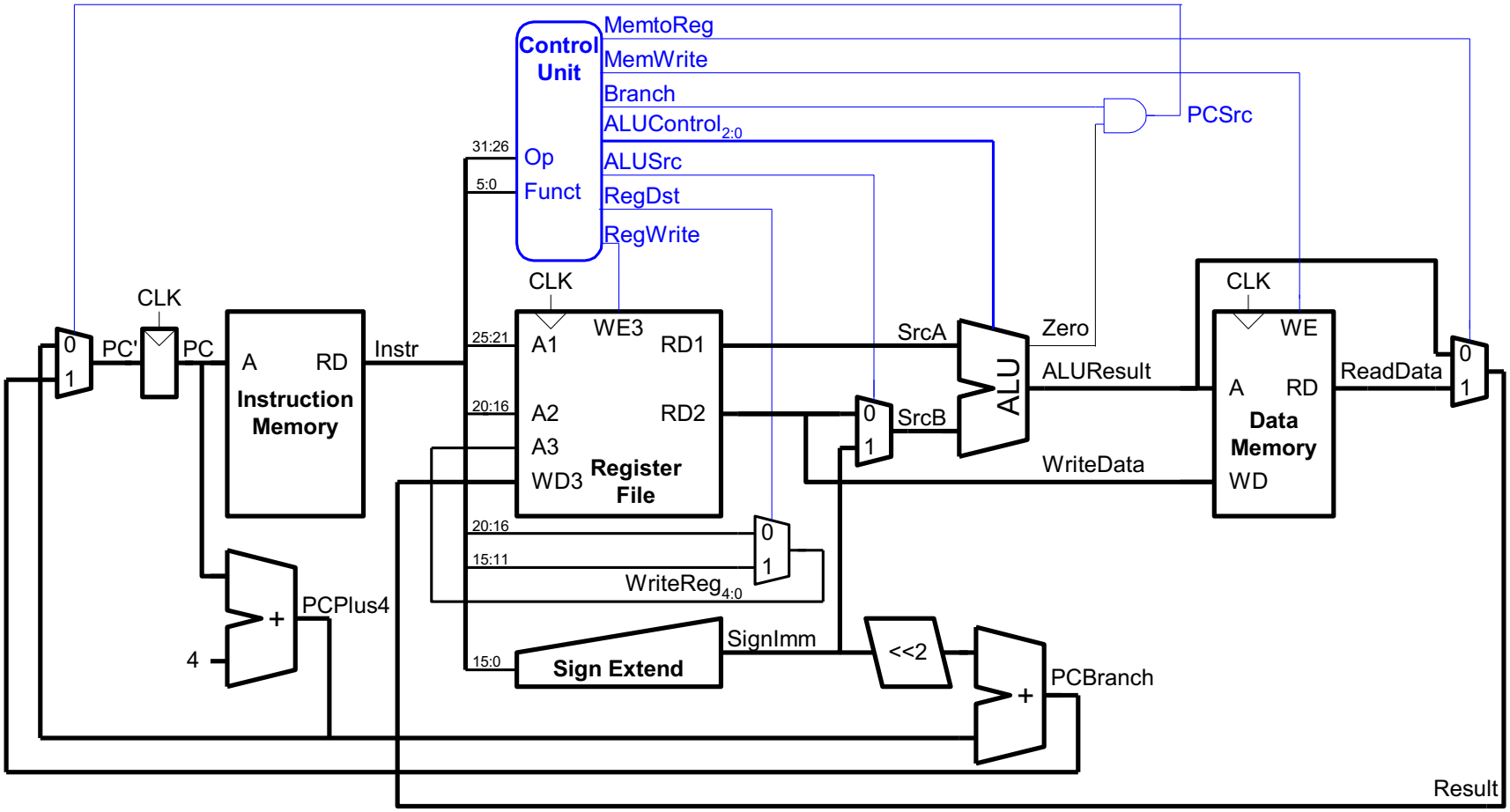
# Extended Functionality: j



# Control Unit: Main Decoder

| Instruction | $Op_{5:0}$ | RegWrite | RegDst | AluSrc | Branch | MemWrite | MemtoReg | $ALUOp_{1:0}$ | Jump |
|---|---|---|---|---|---|---|---|---|---|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 | 0 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 | 0 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 | 0 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 | 0 |
| j | 000100 | 0 | X | X | X | 0 | X | XX | 1 |

# Review: Complete Single-Cycle Processor (H&H)



# A Bit More on Performance Analysis

# Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

# Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.
- **So how fast are my instructions ?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

# Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.
- **So how fast are my instructions ?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***
- **How much time is one clock cycle?**
  - The critical path determines how much time one cycle requires = ***clock period***.
  - 1/clock period = ***clock frequency*** = how many cycles can be done each second.

# Performance Analysis

- Execution time of an instruction
  - {CPI} x {clock cycle time}

- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

# Processor Performance

- **Now as a general formula**
  - Our program consists of executing **N** instructions.
  - Our processor needs **CPI** cycles for each instruction.
  - The maximum clock speed of the processor is **f**, and the clock period is therefore **T**=1/f

# Processor Performance

- **Now as a general formula**
  - Our program consists of executing **N** instructions.
  - Our processor needs **CPI** cycles for each instruction.
  - The maximum clock speed of the processor is **f**, and the clock period is therefore **T**=1/f
- **Our program will execute in**

$$N \times CPI \times (1/f) = N \times CPI \times T \text{ seconds}$$

# How can I Make the Program Run Faster?

**N x CPI x (1/f)**

# How can I Make the Program Run Faster?

## N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers

# How can I Make the Program Run Faster?

## N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel

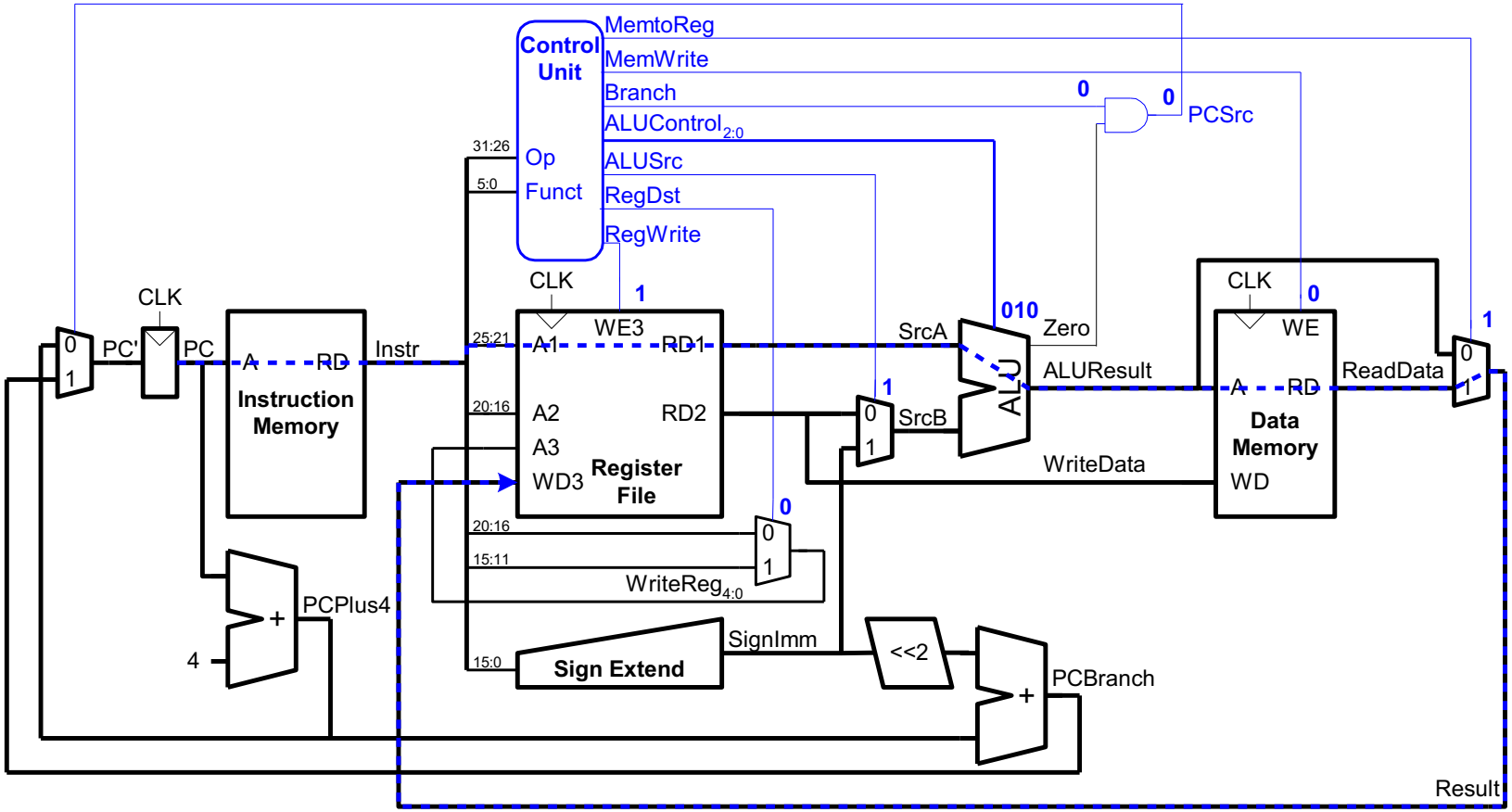
# How can I Make the Program Run Faster?

## N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel
- **Increase the clock frequency**
  - Find a ‘newer’ technology to manufacture
  - Redesign time critical components
  - Adopt pipelining

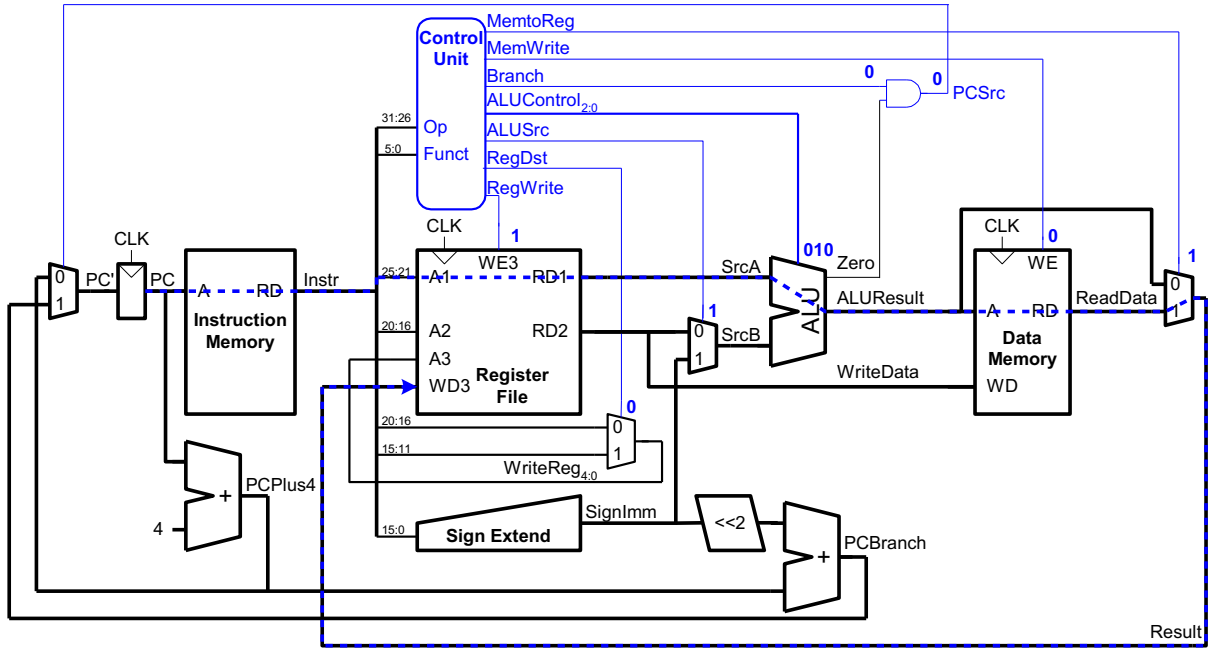
# Single-Cycle Performance

- $T_C$ is limited by the critical path (`lw`)



# Single-Cycle Performance

- **Single-cycle critical path:**
  - $T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$
- **In most implementations, limiting paths are:**
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$



# Single-Cycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_PC}$ | 30 |
| Register setup | $t_{setup}$ | 20 |
| Multiplexer | $t_{mux}$ | 25 |
| ALU | $t_{ALU}$ | 200 |
| Memory read | $t_{mem}$ | 250 |
| Register file read | $t_{RFread}$ | 150 |
| Register file setup | $t_{RFsetup}$ | 20 |

$T_c$ =

# Single-Cycle Performance Example

| Element | Parameter | Delay (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_PC}$ | 30 |
| Register setup | $t_{setup}$ | 20 |
| Multiplexer | $t_{mux}$ | 25 |
| ALU | $t_{ALU}$ | 200 |
| Memory read | $t_{mem}$ | 250 |
| Register file read | $t_{RFread}$ | 150 |
| Register file setup | $t_{RFsetup}$ | 20 |

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$$
$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$
$$= 925 \text{ ps}$$

# Single-Cycle Performance Example

- **Example:**

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

# Single-Cycle Performance Example

- **Example:**

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

***Execution Time*** = # instructions x CPI x TC

= $(100 \times 10^{9})(1)(925 \times 10^{-12}$ s)

= 92.5 seconds